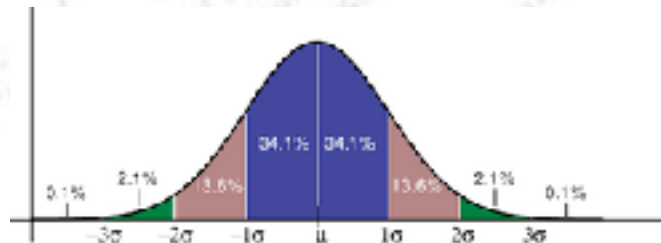


Machine Learning

Lecture 1: BDTs, NNs and ML in HEP



Troels C. Petersen (NBI)



"Statistics is merely a quantisation of common sense - Machine Learning is a sharpening of it!"

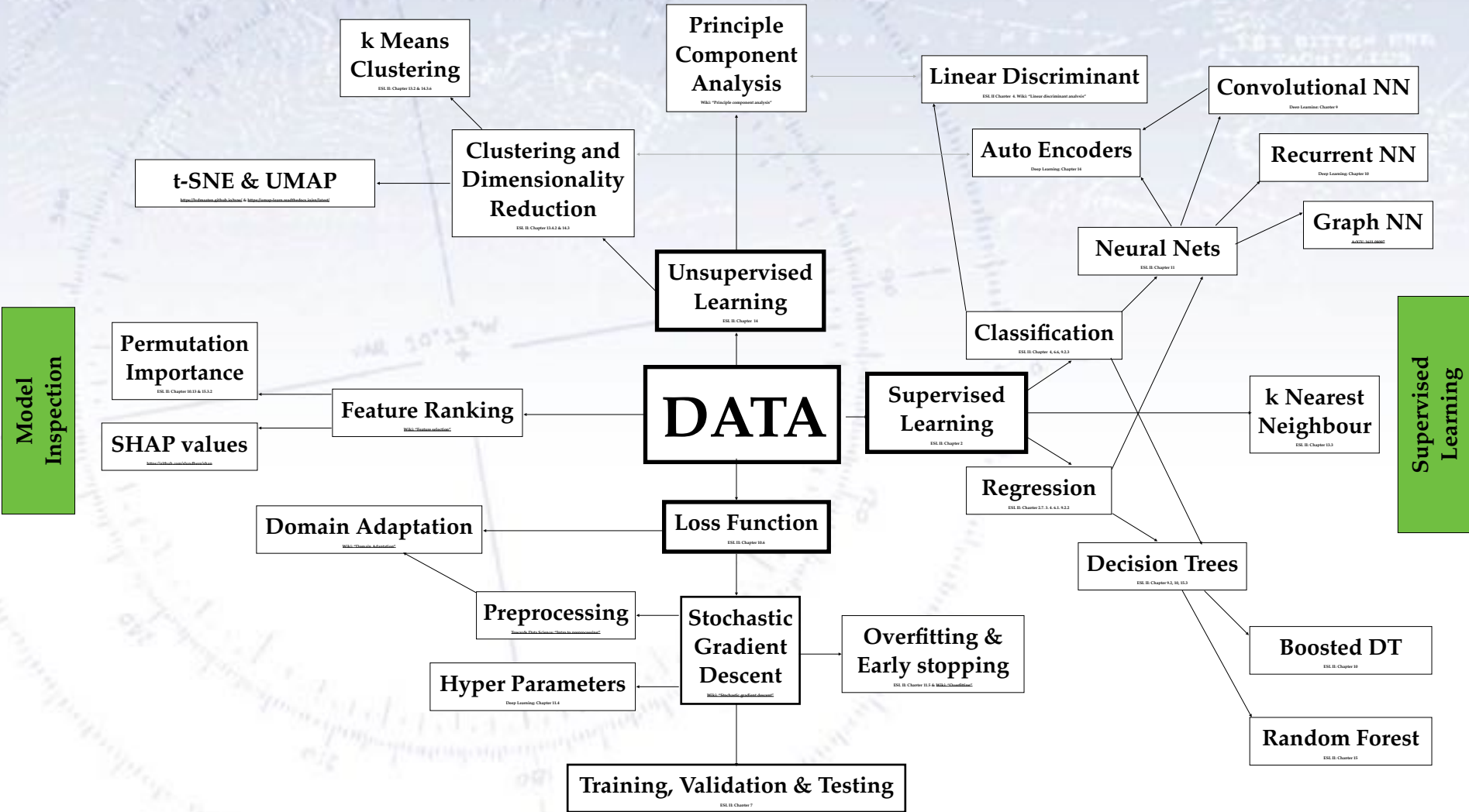
Comment on “The AI Hype”

Machine Learning is a tool like all others (logic, math, computers, statistics, etc.)

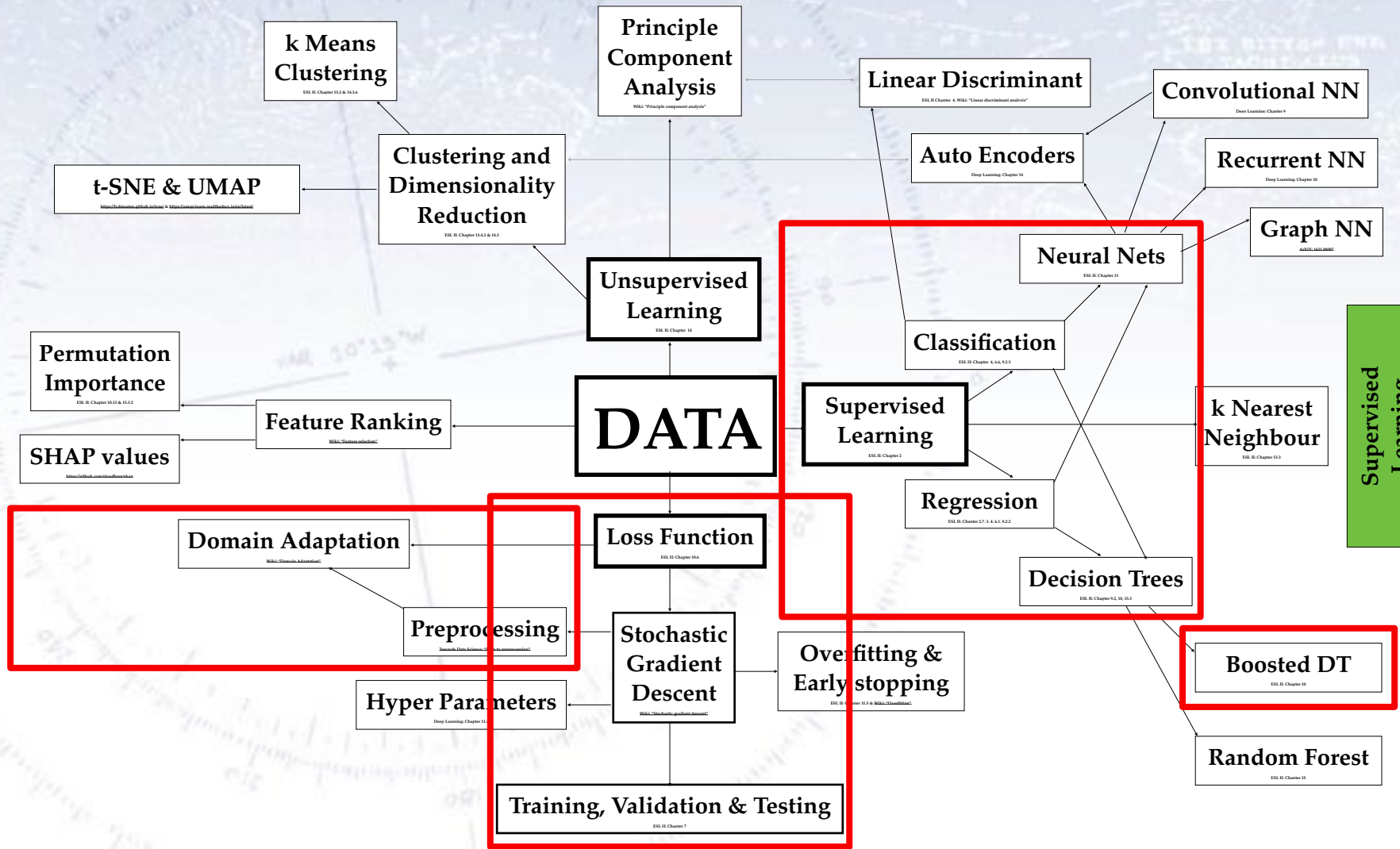
Despite the connotations of machine learning and artificial intelligence as a mysterious and radical departure from traditional approaches, we stress that machine learning has a mathematical formulation that is closely tied to statistics, the calculus of variations, approximation theory, and optimal control theory.

[PDG 2024, Review of Machine Learning]

So this is just a sharpening of our tools... albeit a cool sharpening!



References:
Trevor Hastie et al.: "Elements of Statistics Learning II" (ELS II)
Ian Goodfellow et al.: "Deep Learning"
Wiki: Good reference for most subjects (only specified when essential)
Various blogs / githubs / papers for specific subjects.



References:
Trevor Hastie et al.: "Elements of Statistics Learning II" (ELS II)
Ian Goodfellow et al.: "Deep Learning"
Wiki: Good reference for most subjects (only specified when essential)
Various blogs / githubs / papers for specific subjects.

Lecture 1: BDTs, NNs & ML in HEP

Outline

What is ML & Humans vs. ML

Two main ingredients:

- Universal Approximation Theorems
- Stochastic Gradient Descent

The linear vs. non-linear case

Tree based models

Neural Network models

Loss functions

Train, Validation & Test

Preprocessing

Domain Adaptation:

- What are the dangers?
- MC signal, data background
- How to find data-MC differences
- How to mend data-MC differences
- Training in/with data

Coding example



What is ML?

What is Machine Learning?

While there is no formal definition, an early attempt is the following intuition:

“Machine learning programs can perform tasks without being explicitly programmed to do so.”

[Arthur Samuel, US computer pioneer 1901-1990]

“Little Peter is capable of finding his way home without being explicitly taught to do so.”

What is Machine Learning?

While there is no formal definition, an early attempt is the following intuition:

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T , as measured by P , improves with experience E ."

[T. Mitchell, "Machine Learning" 1997]

"Little Peter is said to learn from traveling around with respect to finding his way home and the time it takes, if his ability to find his way home, as measured by the time it takes, improves as he travels around."



Humans vs. ML

Dimensionality and Complexity

Humans are good at seeing/understanding data in few dimensions!

However, as dimensionality grows, complexity grows exponentially (“curse of dimensionality”), and humans are generally not geared for such challenges.

	Low dim.	High dim.
Linear	Humans: Computers:	Humans: Computers:
Non-linear	Humans: Computers:	Humans: Computers:

Computers, on the other hand, are OK with high dimensionality, albeit the growth of the challenge, but have a harder time facing non-linear issues.

However, through smart algorithms, computers have learned to deal with it all!

That is essentially what Machine Learning has enabled!

Dimensionality and Complexity

Humans are good at seeing/understanding data in few dimensions!
However, as dimensionality grows, complexity grows exponentially (“curse of dimensionality”), and humans are generally not geared for such challenges.

	Low dim.	High dim.
Linear	Humans: Computers:	Humans: Computers:
Non-linear	Humans: Computers:	Humans: Computers:

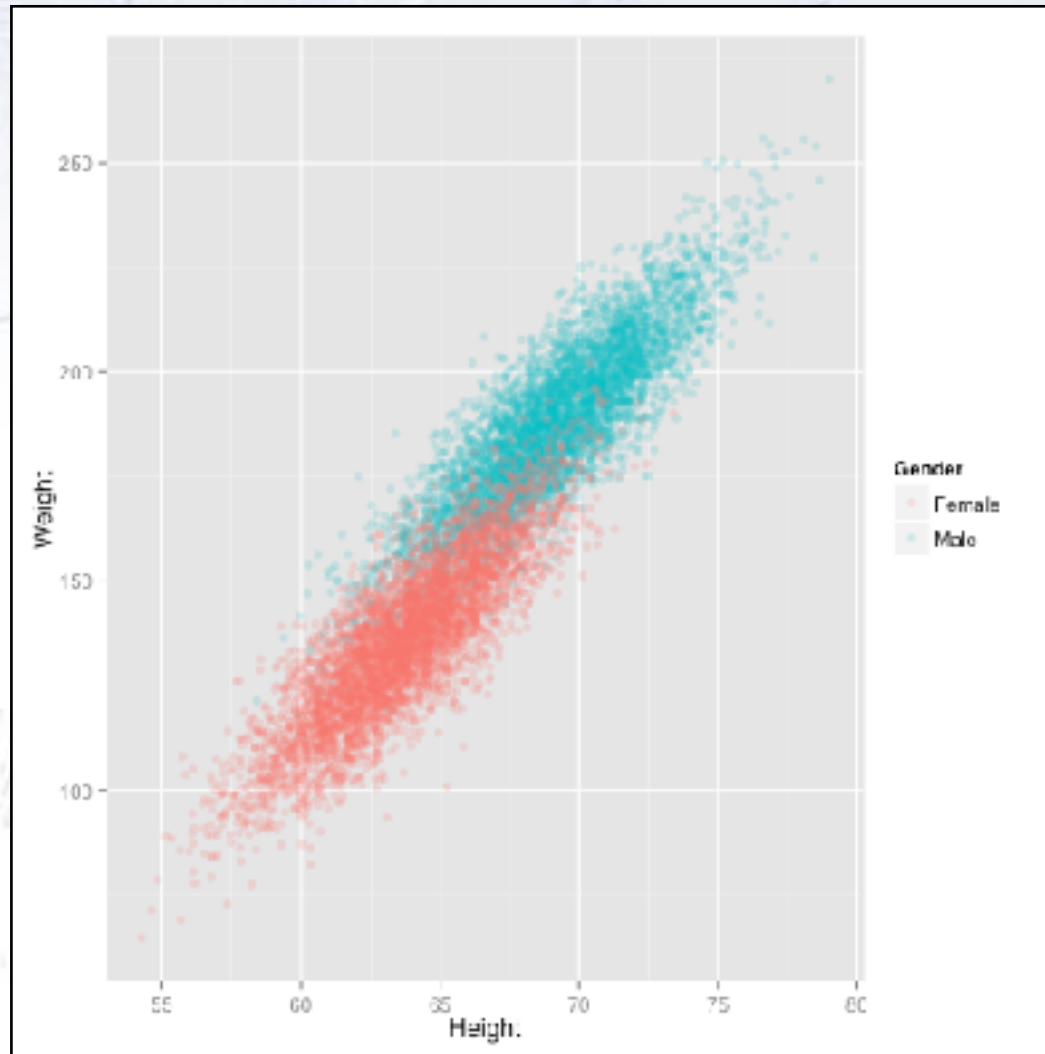
Computers, on the other hand, are OK with high dimensionality, albeit the growth of the challenge, but have a harder time facing non-linear issues.

However, through smart algorithms, computers have learned to deal with it all!

That is essentially what Machine Learning has enabled!

Dimensionality and Complexity

Humans & Computers are good at seeing/understanding linear data in few dimensions:



Dimensionality and Complexity

Humans are good at seeing/understanding data in few dimensions!
However, as dimensionality grows, complexity grows exponentially (“curse of dimensionality”), and humans are generally not geared for such challenges.

	Low dim.	High dim.
Linear	Humans: ✓ Computers: ✓	Humans: Computers:
Non-linear	Humans: Computers:	Humans: Computers:

Computers, on the other hand, are OK with high dimensionality, albeit the growth of the challenge, but have a harder time facing non-linear issues.

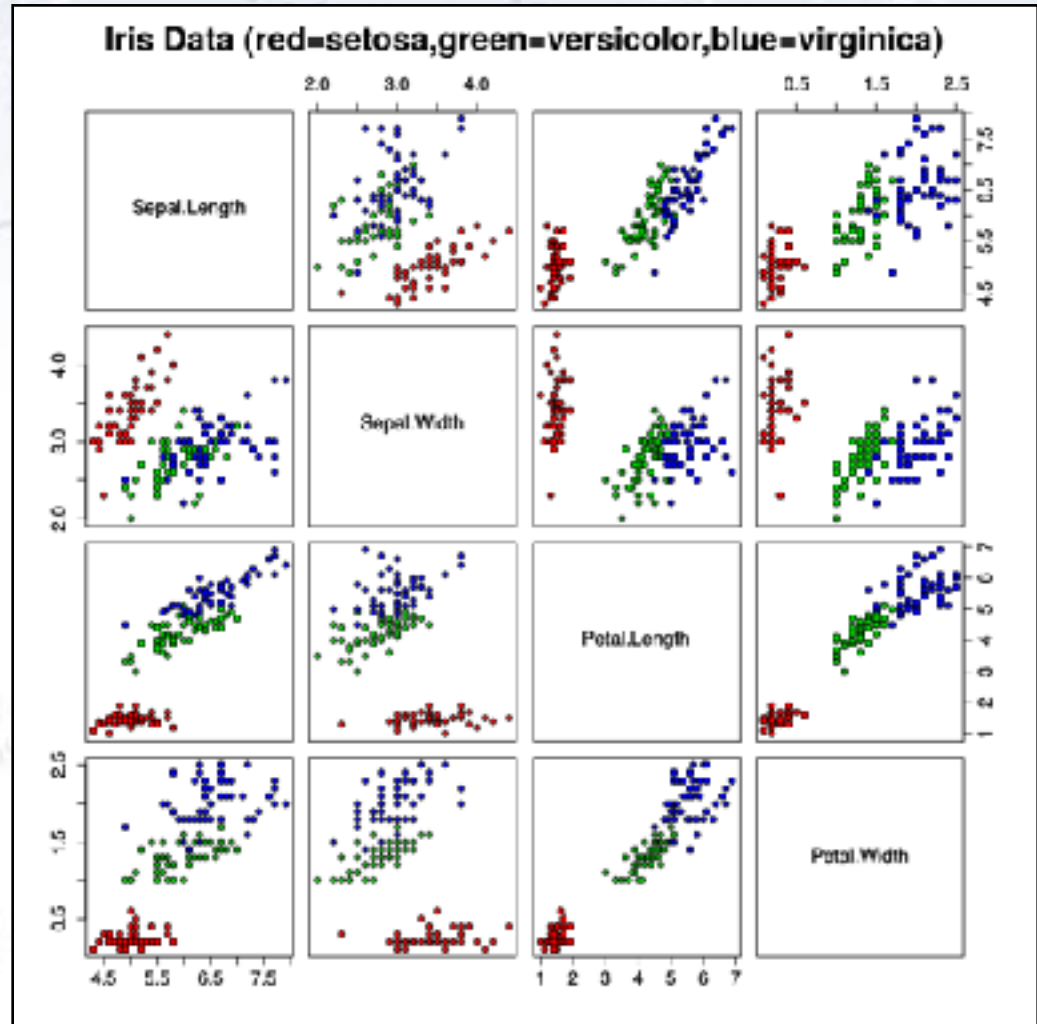
However, through smart algorithms, computers have learned to deal with it all!
That is essentially what Machine Learning has enabled!

Dimensionality and Complexity

However, when the dimensionality goes beyond 3D, we are lost, even for simple linear data. Computers are not...

Shown is the famous Fisher Iris dataset:
150 irises (3 kinds) with
4 measurements for each.

4 dimensional data!



Dimensionality and Complexity

Humans are good at seeing/understanding data in few dimensions!

However, as dimensionality grows, complexity grows exponentially (“curse of dimensionality”), and humans are generally not geared for such challenges.

	Low dim.	High dim.
Linear	Humans: ✓ Computers: ✓	Humans: ÷ Computers: ✓
Non-linear	Humans: Computers:	Humans: Computers:

Computers, on the other hand, are OK with high dimensionality, albeit the growth of the challenge, but have a harder time facing non-linear issues.

However, through smart algorithms, computers have learned to deal with it all!

That is essentially what Machine Learning has enabled!



Jackson Pollock

Dimensionality and Complexity

Humans are good at seeing/understanding data in few dimensions!

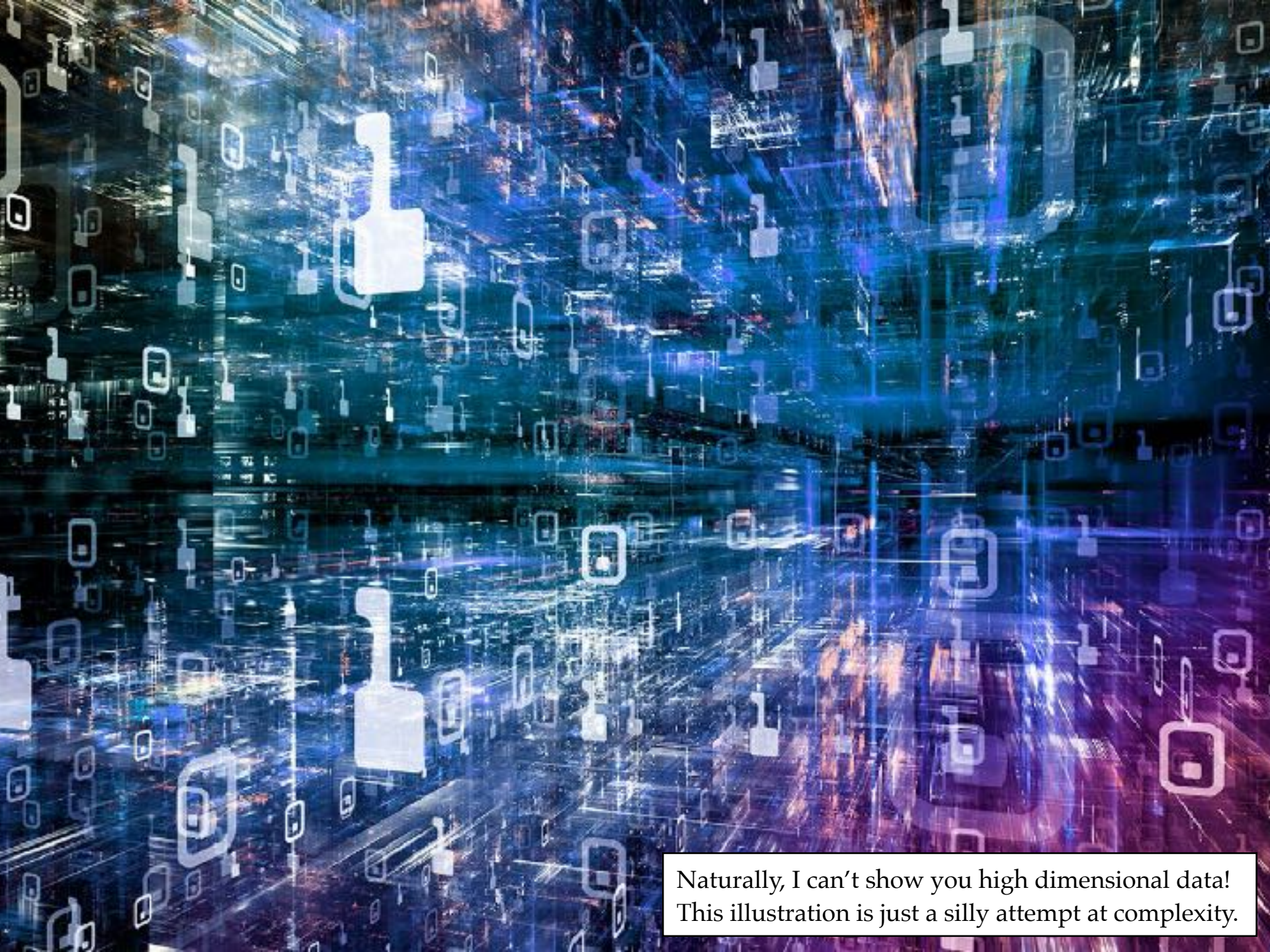
However, as dimensionality grows, complexity grows exponentially (“curse of dimensionality”), and humans are generally not geared for such challenges.

	Low dim.	High dim.
Linear	Humans: ✓ Computers: ✓	Humans: ÷ Computers: ✓
Non-linear	Humans: ✓ Computers: (✓)	Humans: Computers:

Computers, on the other hand, are OK with high dimensionality, albeit the growth of the challenge, but have a harder time facing non-linear issues.

However, through smart algorithms, computers have learned to deal with it all!

That is essentially what Machine Learning has enabled!



Naturally, I can't show you high dimensional data!
This illustration is just a silly attempt at complexity.

Dimensionality and Complexity

Humans are good at seeing/understanding data in few dimensions!

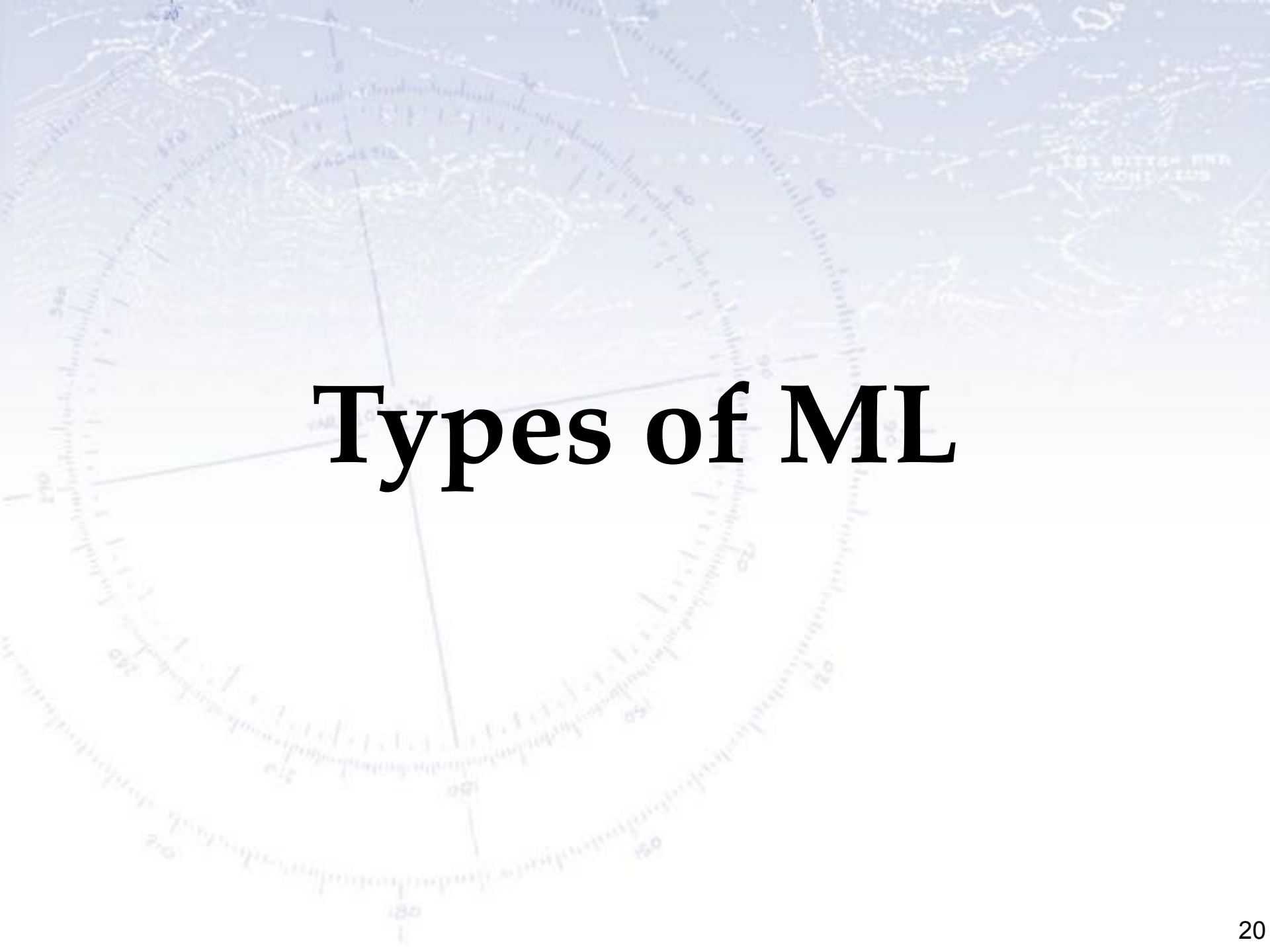
However, as dimensionality grows, complexity grows exponentially (“curse of dimensionality”), and humans are generally not geared for such challenges.

	Low dim.	High dim.
Linear	Humans: ✓ Computers: ✓	Humans: ÷ Computers: ✓
Non-linear	Humans: ✓ Computers: (✓)	Humans: ÷ Computers: (✓)

Computers, on the other hand, are OK with high dimensionality, albeit the growth of the challenge, but have a harder time facing non-linear issues.

However, through smart algorithms, computers have learned to deal with it all!

That is essentially what Machine Learning has enabled!



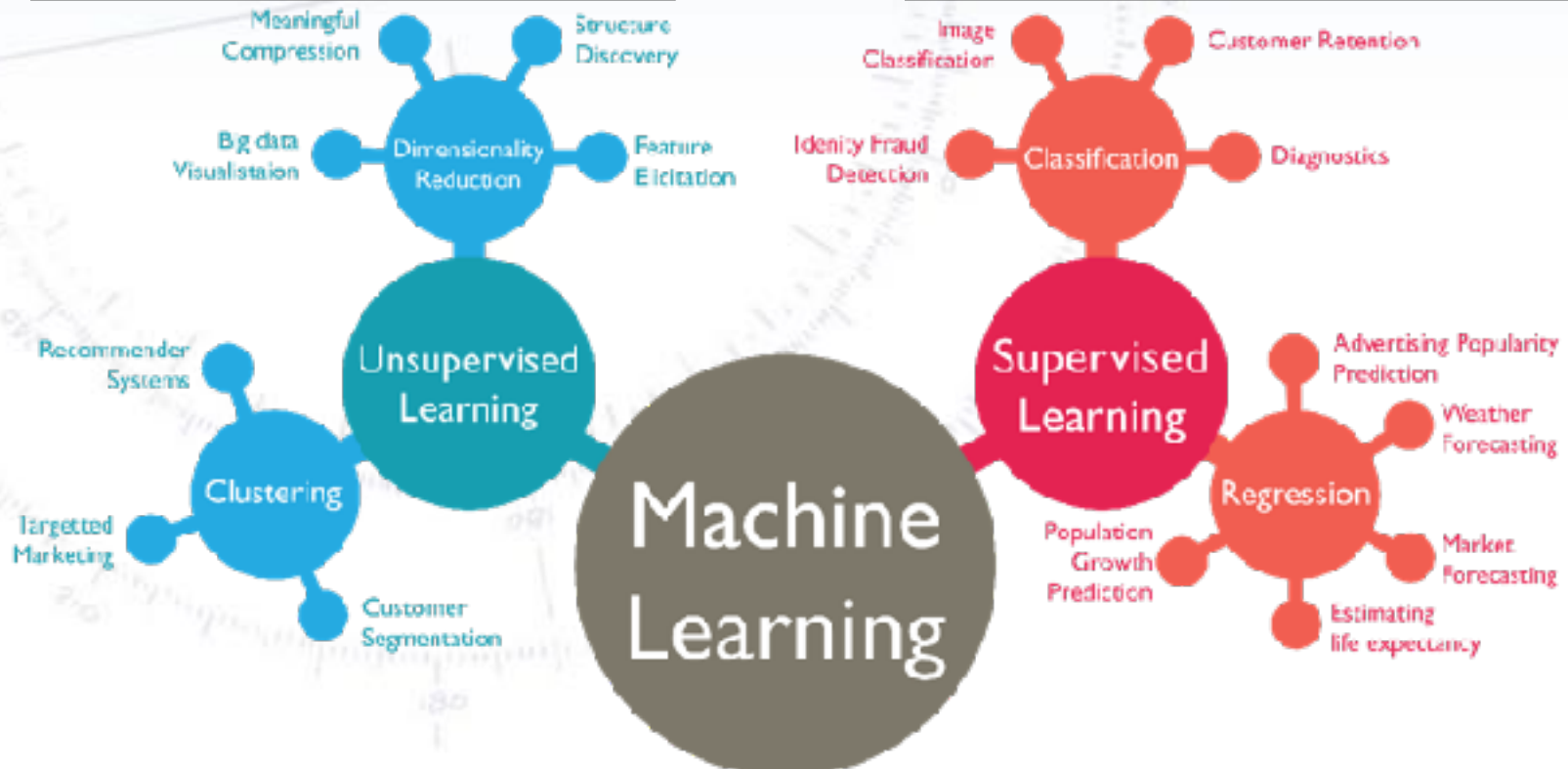
Types of ML

Unsupervised vs. Supervised Classification vs. Regression

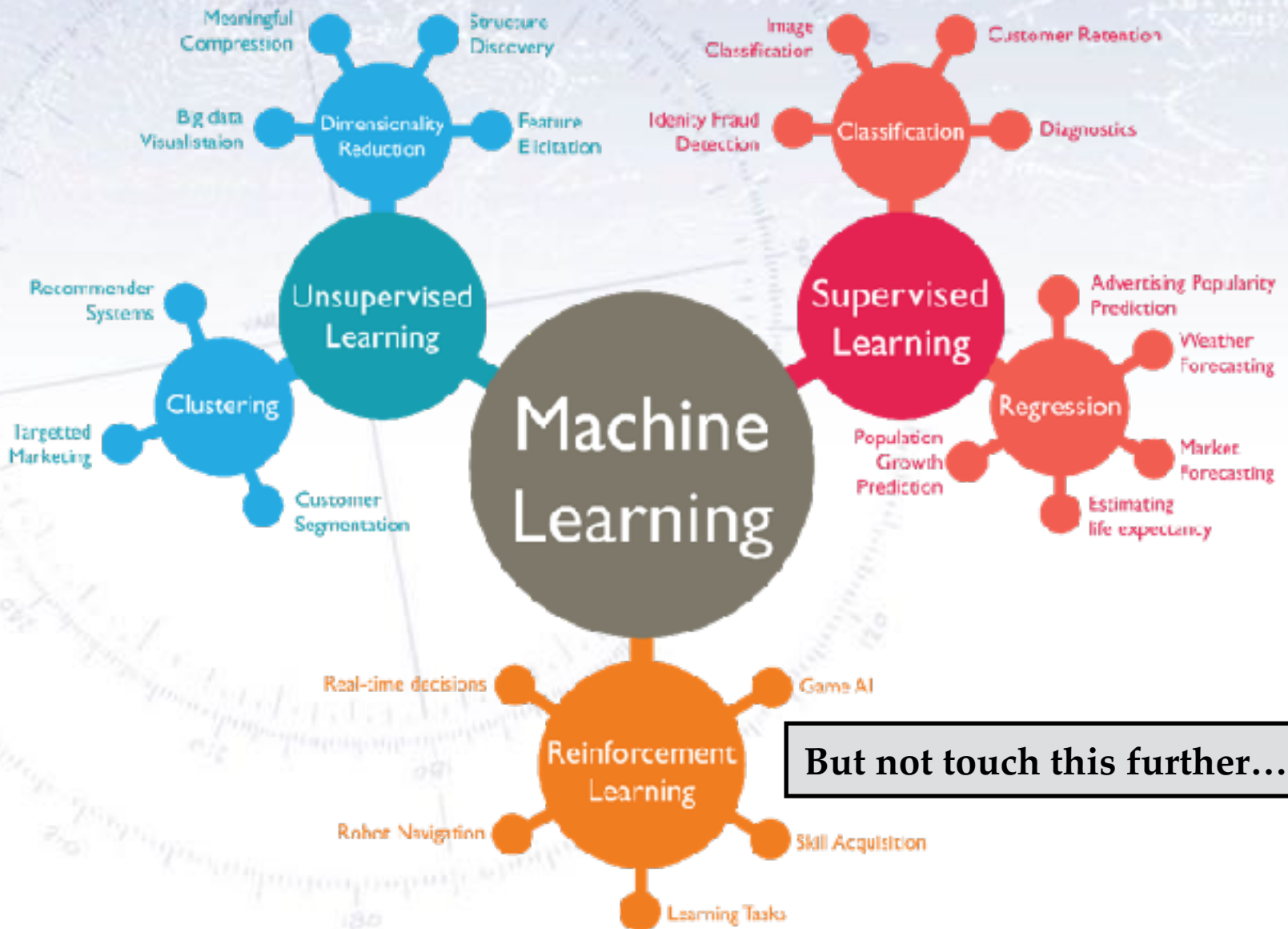
Machine Learning can be supervised (you have correctly labelled examples) or unsupervised (you don't)... [or reinforced]. Following this, one can be using ML to either classify (is it A or B?) or for regression (estimate of X).

But of course also over here!

We will be mostly on this side!



Reinforcement Learning



But not touch this further...



Two main ingredients:

1. Solutions exists

2. How to find them

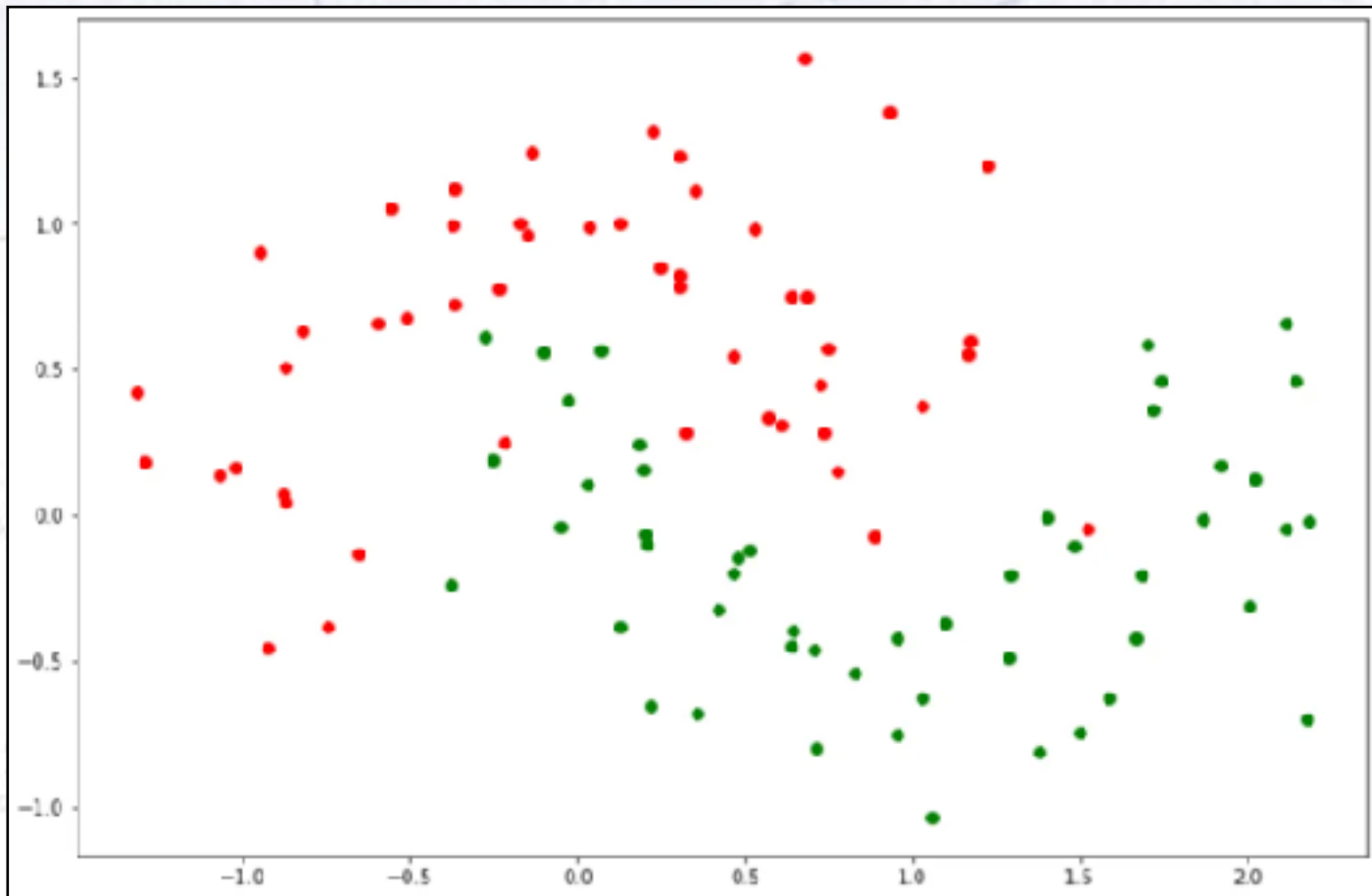
A background map of the Pacific Ocean, showing a path highlighted in yellow. The path starts near the top center, moves down, then curves to the right, then down again, and finally curves to the left. The map includes latitude and longitude lines, and some text labels like 'WASH DC' and '181 BITTCH ERB TACHI / 129'.

Solutions exists

(Technically called Universal Approximation Theorems)

Where to separate?

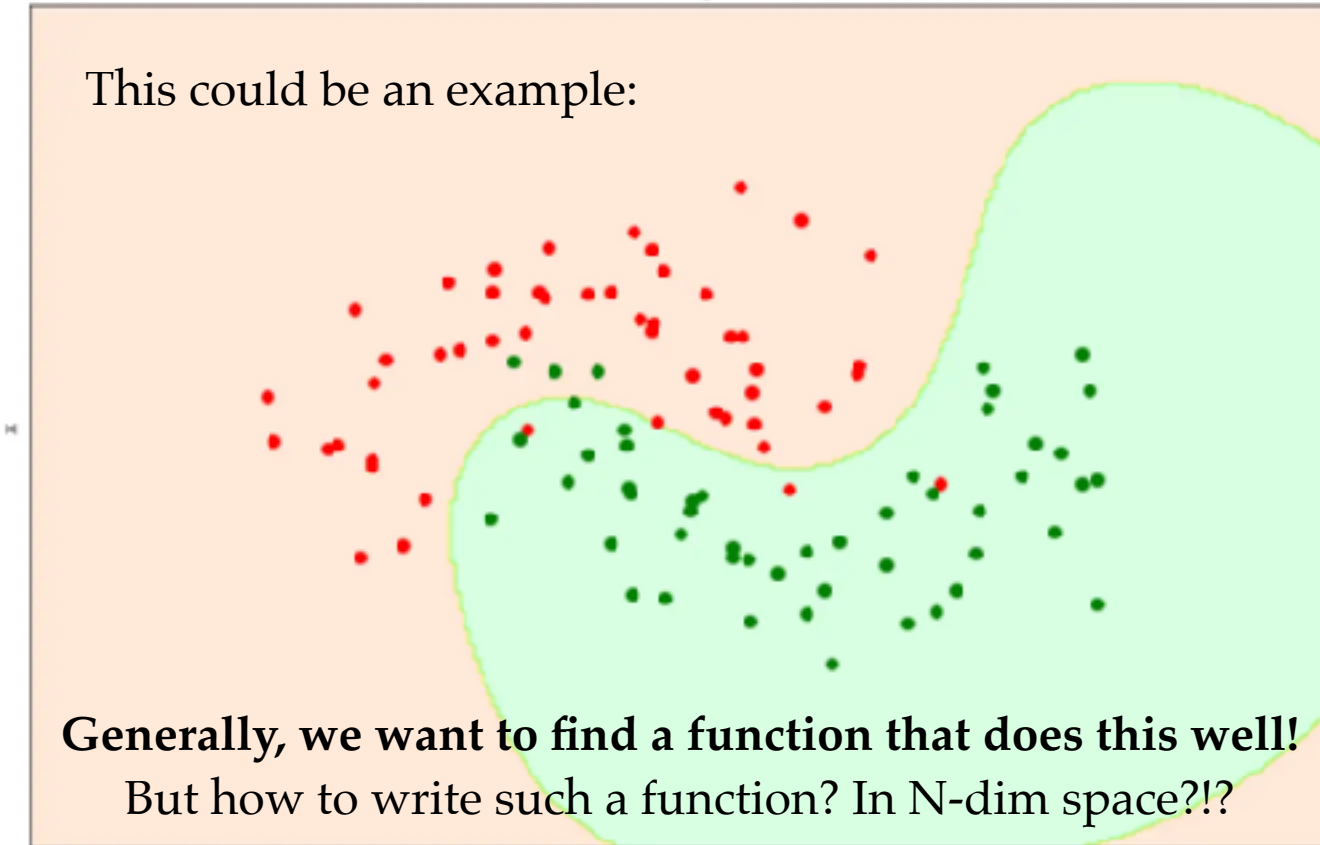
Look at the red and green points, and imagine that you wanted to draw a curve that separates these.



Where to separate?

Look at the red and green points, and imagine that you wanted to draw a curve that separates these.

This could be an example:



Generally, we want to find a function that does this well!
But how to write such a function? In N-dim space?!?

Universal Approx. Theorems

A simple function can be obtained simply by asking a lot of questions:

Question: Is $B > 0.23$?

Answer: Yes \rightarrow Red

Answer: No \rightarrow Blue

This question is illustrated in the drawing by the horizontal line with red and blue on the sides.

A Decision Tree consists of asking many such questions, corresponding to setting a lot of lines.



Universal Approx. Theorems

A simple function can be obtained simply by asking a lot of questions:

Question: Is $B > 0.23$?

Answer: Yes \rightarrow Red

Answer: No \rightarrow Blue

This question is illustrated in the drawing by the horizontal line with red and blue on the sides.

A Decision Tree consists of asking many such questions, corresponding to setting a lot of lines.



Universal Approx. Theorems

A simple function can be obtained simply by asking a lot of questions:

Question: Is $B > 0.23$?

Answer: Yes \rightarrow Red

Answer: No \rightarrow Blue

This question is illustrated in the drawing by the horizontal line with red and blue on the sides.

A Decision Tree consists of asking many such questions, corresponding to setting a lot of lines.



Universal Approx. Theorems

A simple function can be obtained simply by asking a lot of questions:

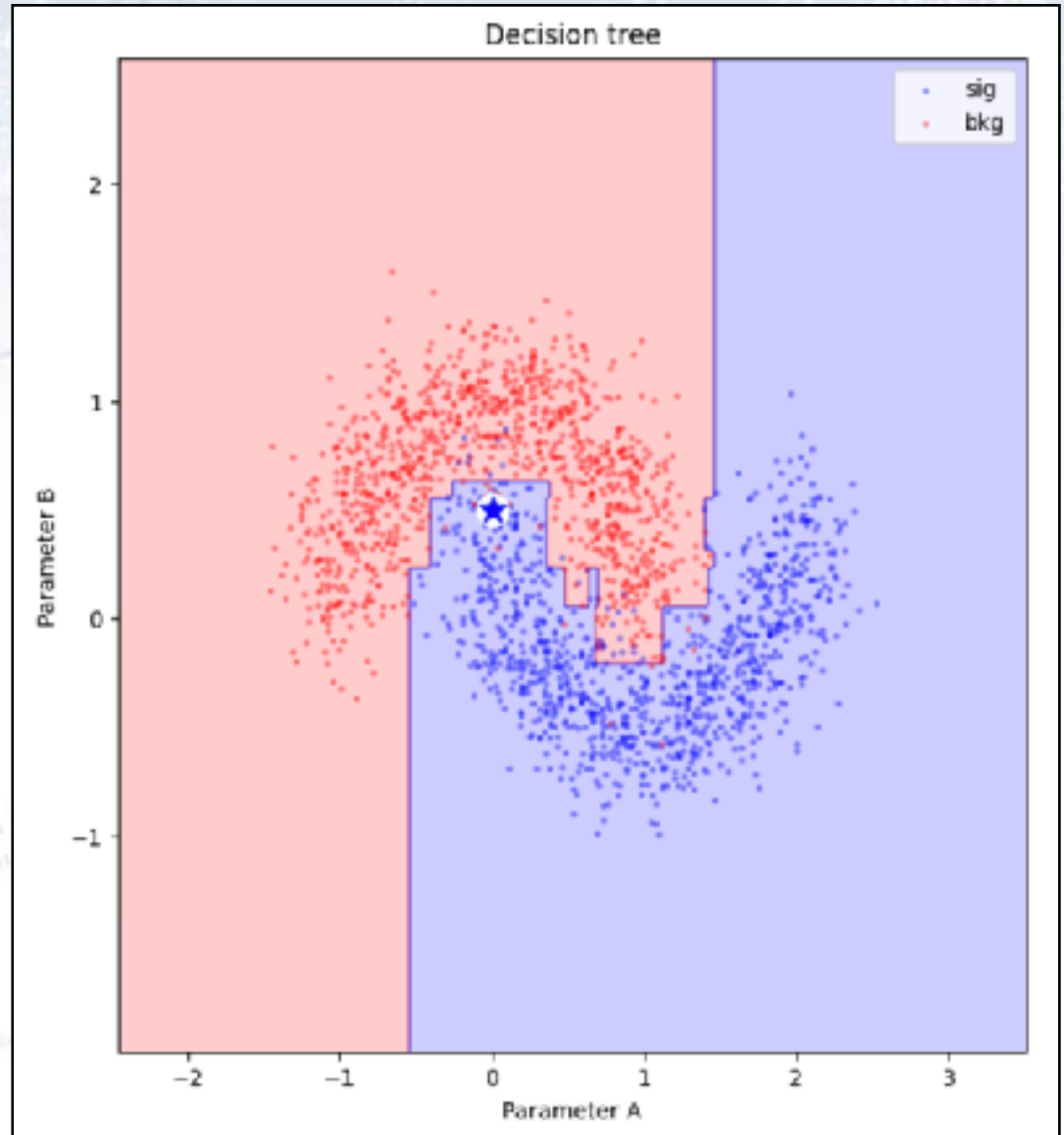
Question: Is $B > 0.23$?

Answer: Yes \rightarrow Red

Answer: No \rightarrow Blue

This question is illustrated in the drawing by the horizontal line with red and blue on the sides.

A Decision Tree consists of asking many such questions, corresponding to setting a lot of lines.



Universal Approx. Theorems

A simple function can be obtained simply by asking a lot of questions:

Question: Is $B > 0.23$?

Answer: Yes \rightarrow Red

Answer: No \rightarrow Blue

This question is illustrated in the drawing by the horizontal line with red and blue on the sides.

A Decision Tree consists of asking many such questions, corresponding to setting a lot of lines.



Universal Approx. Theorems

A simple function can be obtained simply by asking a lot of questions:

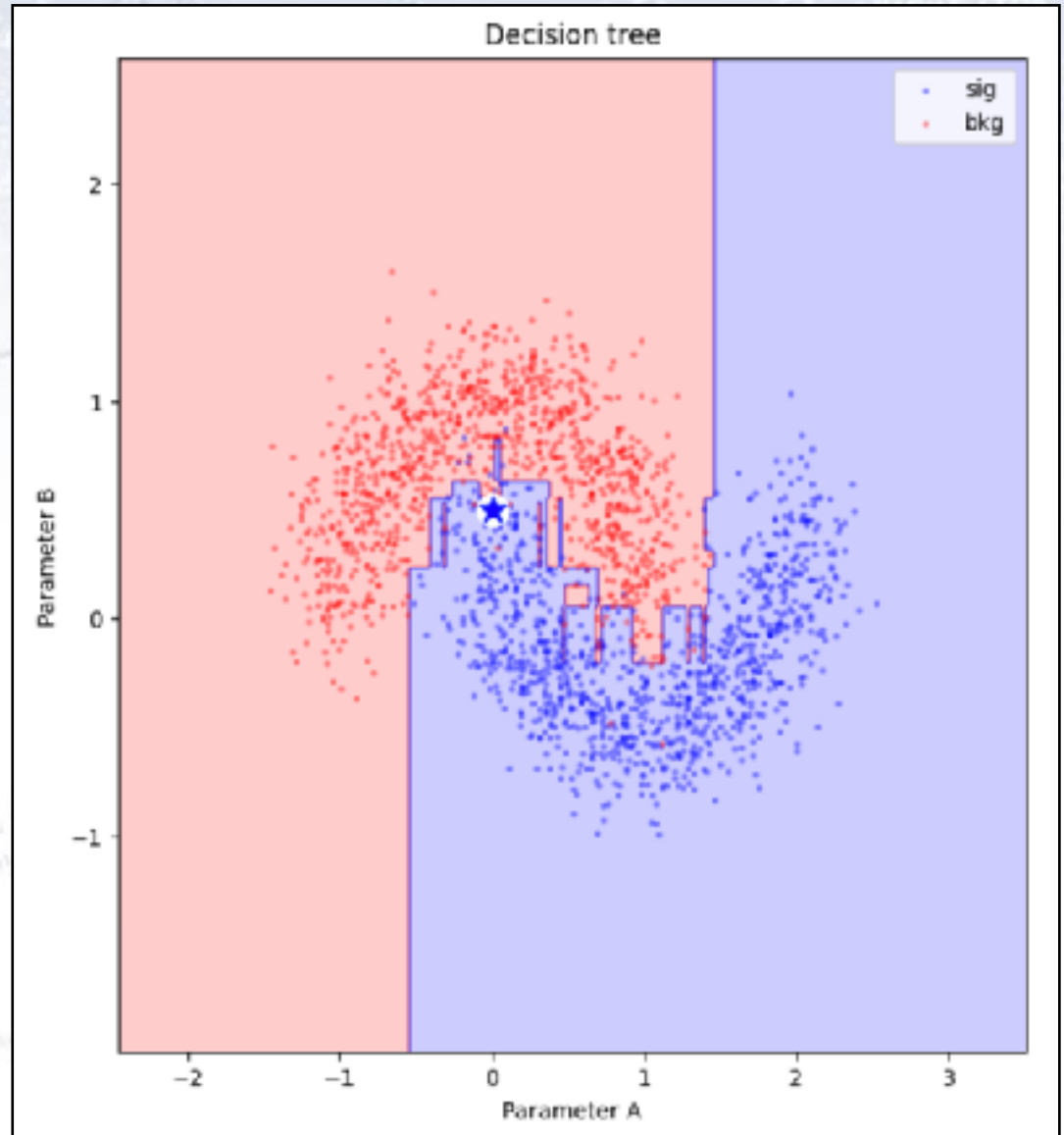
Question: Is $B > 0.23$?

Answer: Yes \rightarrow Red

Answer: No \rightarrow Blue

This question is illustrated in the drawing by the horizontal line with red and blue on the sides.

A Decision Tree consists of asking many such questions, corresponding to setting a lot of lines.



Universal Approximation Theorems

Theorem 5.1.1 (Universal Approximation Theorem) ¹⁰ Let σ be a non-constant, bounded, and monotone-increasing continuous function. Let I_{m_0} denote the m_0 -dimensional unit hypercube $[0, 1]^{m_0}$. The space of continuous functions on I_{m_0} is denoted as $C(I_{m_0})$. Then given any function $f \in C(I_{m_0})$ and $\epsilon > 0$ there exists a set of real constants a_i, b_i and w_{ij} , where $i = 1, \dots, m_1$ and $j = 1, \dots, m_0$ such that we may define

$$F(x_1, \dots, x_{m_0}) = \sum_{i=1}^{m_1} a_i \sigma \left(\sum_{j=1}^{m_0} w_{ij} x_j + b_i \right) \quad (5.6)$$

as an approximate realization of the function f ; that is,

$$|F(x_1, \dots, x_{m_0}) - f(x_1, \dots, x_{m_0})| < \epsilon \quad (5.7)$$

for all x_1, x_2, \dots, x_{m_0} that lie in the input space.

Universal Approximation Theorems

Theorem 5.1.1 (Universal Approximation Theorem) ¹⁰ Let σ be a non-

Summary:

Neural Networks etc. can approximate functions in any dimension very well!

$$F(x_1, \dots, x_{m_0}) = \sum_{i=1} a_i \sigma \left(\sum_{j=1} w_{ij} x_j + b_i \right) \quad (5.6)$$

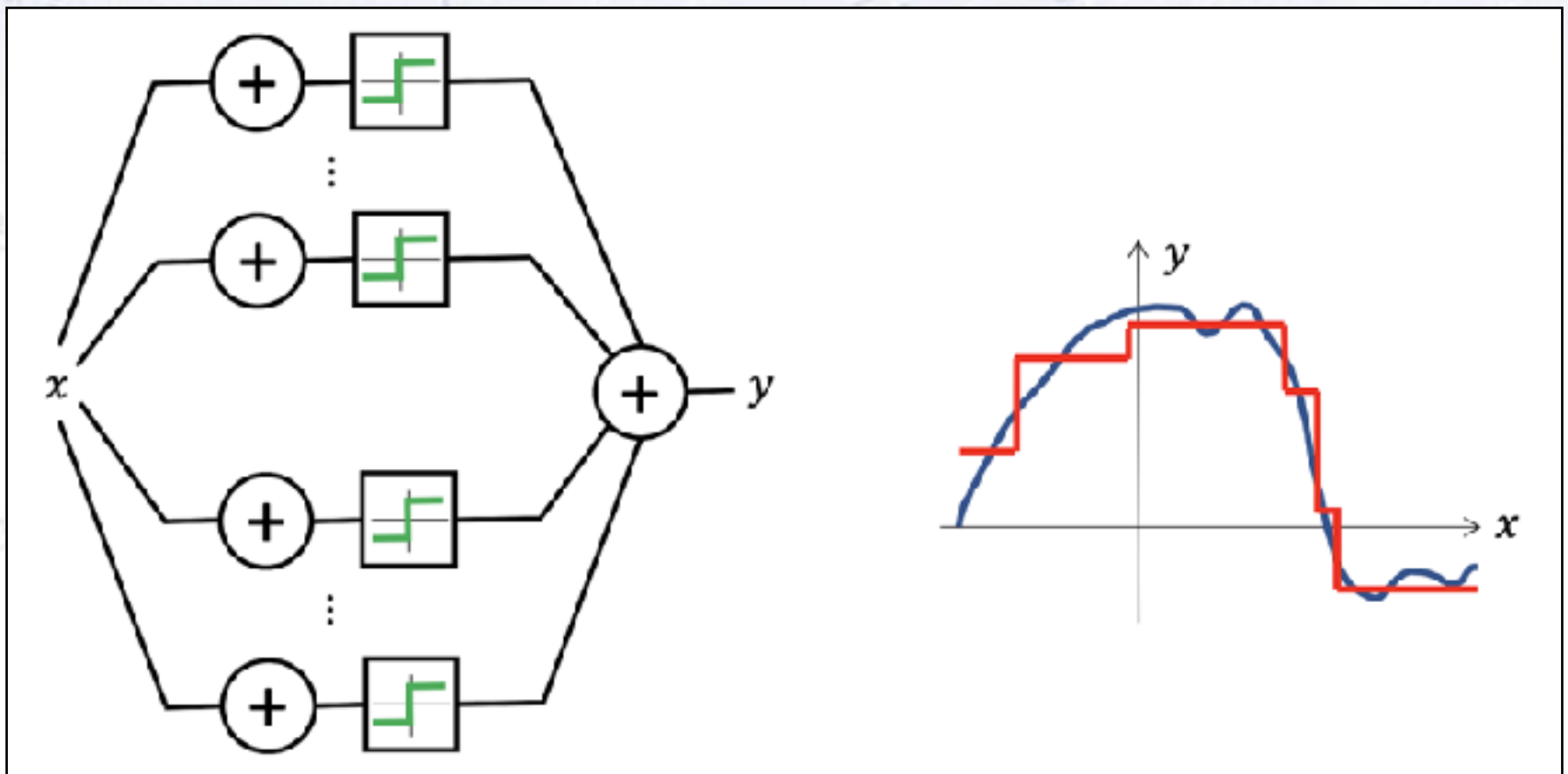
as an approximate realization of the function f ; that is,

$$|F(x_1, \dots, x_{m_0}) - f(x_1, \dots, x_{m_0})| < \epsilon \quad (5.7)$$

for all x_1, x_2, \dots, x_{m_0} that lie in the input space.

Universal Approx. Theorems

Such approximations typically entails a **large amount of parameters**, for which the UATs give no recipe on how to find - only that such a construction is possible.



Universal Approx. Theorems

One main ingredient behind ML are **Universal Approximation Theorems (UAT)**.

These imply that Neural Networks can approximate a very wide variety of functions given simple function constraints and enough degrees of freedom.

This typically entails a **large amount of weights**, for which the UATs give **no recipe on how to find** - only that such a construction is possible.

Even if one assumes that there is no noise in the training set, then there will still be **infinitely many functions that passes through all training points** and not all of them will have the same error on an unseen point (i.e. the test set).

So how to find actual solutions that are behaving nicely?

A topographic map of a mountainous region, likely in the Pacific Northwest, showing contour lines and place names like 'WASH STATE' and '181 BITTGM ERG TACHI 1129'. The map is faded and serves as a background for the text.

How to find these

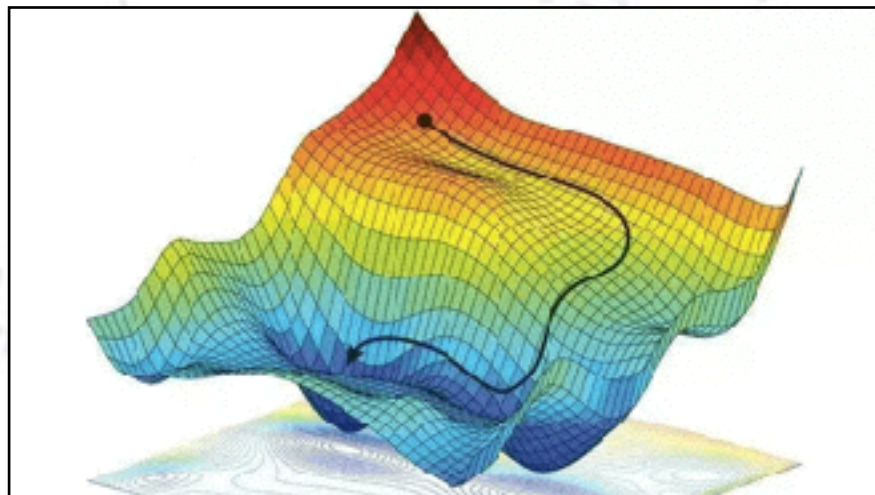
(Technically called Stochastic Gradient Descent)

Stochastic Gradient Descent

The way to obtain the parameters / weights of ML algorithms, is generally by **Stochastic Gradient Descent**.

This “back propagation” algorithm works by computing the gradient of the loss function (to be optimised) with respect to each weight using the chain rule.

One thus computes the gradient one layer at a time, iterating backwards from the last layer (avoiding redundancies). See Goodfellow et al. for details.



(Normal) Gradient Descent

The choice of loss function, L , depends on the problem at hand, and in particular what you find important! You want to minimise this with respect to the model parameters θ :

$$L(\theta) = \frac{1}{N} \sum_i^N L_i(\theta)$$

In order to find the optimal solution, one can use Gradient Descent, typically **based on the whole dataset**:

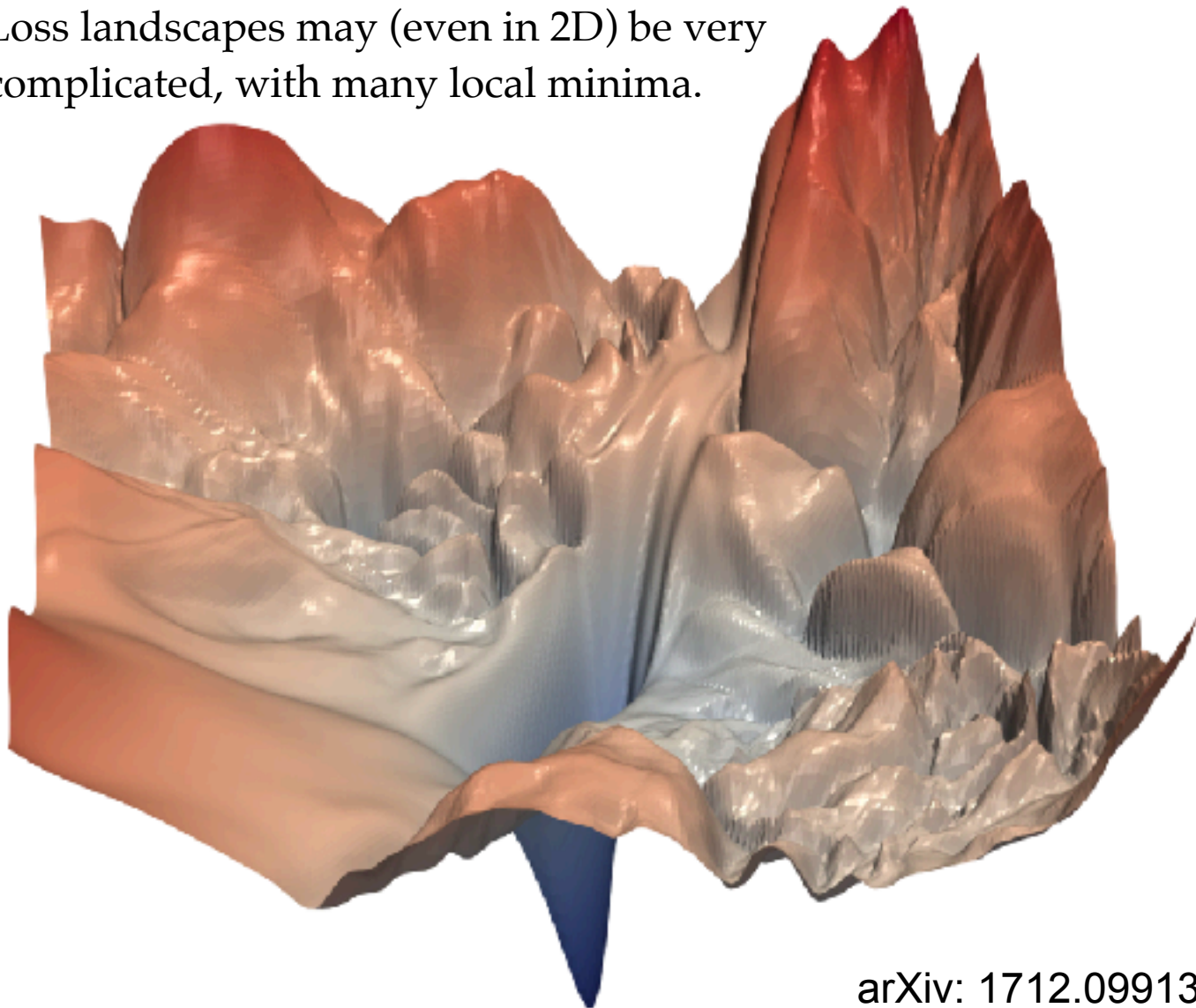
$$\theta_{j+1} = \theta_j - \eta \nabla L(\theta) = \theta_j - \frac{\eta}{N} \sum_i^N \nabla L_i(\theta)$$

This is the procedure used by e.g. Minuit and other minimisation routines.

Note the very important parameter: **Learning rate η** .

(Nasty) Loss Landscapes

Loss landscapes may (even in 2D) be very complicated, with many local minima.



arXiv: 1712.09913

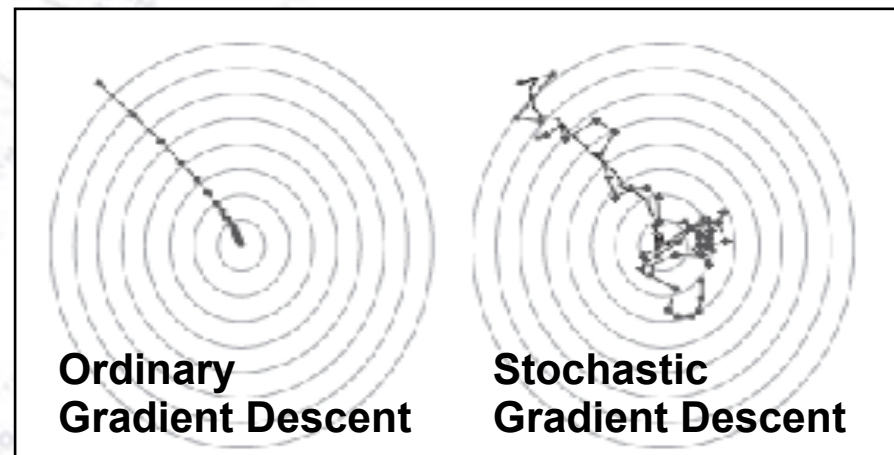
Stochastic Gradient Descent

The way to obtain the parameters / weights of ML algorithms, is generally by **Stochastic Gradient Descent**.

This “back propagation” algorithm works by computing the gradient of the loss function (to be optimised) with respect to each weight using the chain rule.

One thus computes the gradient one layer at a time, iterating backwards from the last layer (avoiding redundancies). See Goodfellow et al. for details.

The gradient descent is made stochastic (and fast) by only considering a fraction (called a “batch”) of the data, when calculating the step in the search for optimal parameters for the algorithm. This allow for stochastic jumping, that avoids local (false) minima.



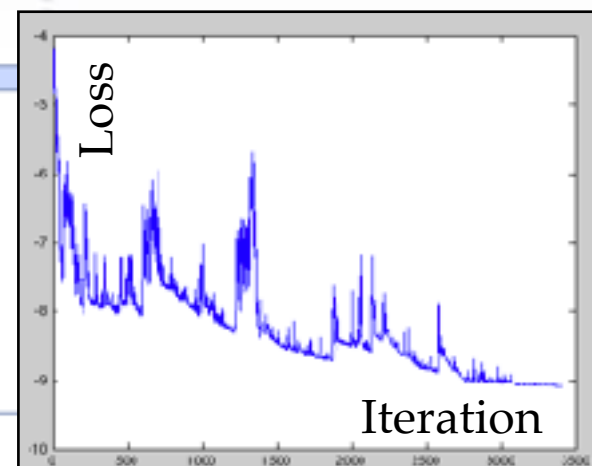
Stochastic Gradient Descent

In order to give the gradient descent some degree of “randomness” (stochastic), one evaluates the below function for **small batches** instead of the full dataset.

$$\theta_{j+1} = \theta_j - \eta \nabla L(\theta) = \theta_j - \frac{\eta}{N} \sum_i^N \nabla L_i(\theta)$$

The algorithm thus becomes:

- Choose an initial vector of parameters w and learning rate η .
- Repeat until an approximate minimum is obtained:
 - Randomly shuffle examples in the training set.
 - For $i = 1, 2, \dots, n$, do:
 - $w := w - \eta \nabla Q_i(w)$.



Not only does this vectorise well and gives smoother descents, but with decreasing learning rate, it “almost surely” finds the global minimum (Robbins-Siegmund theorem).

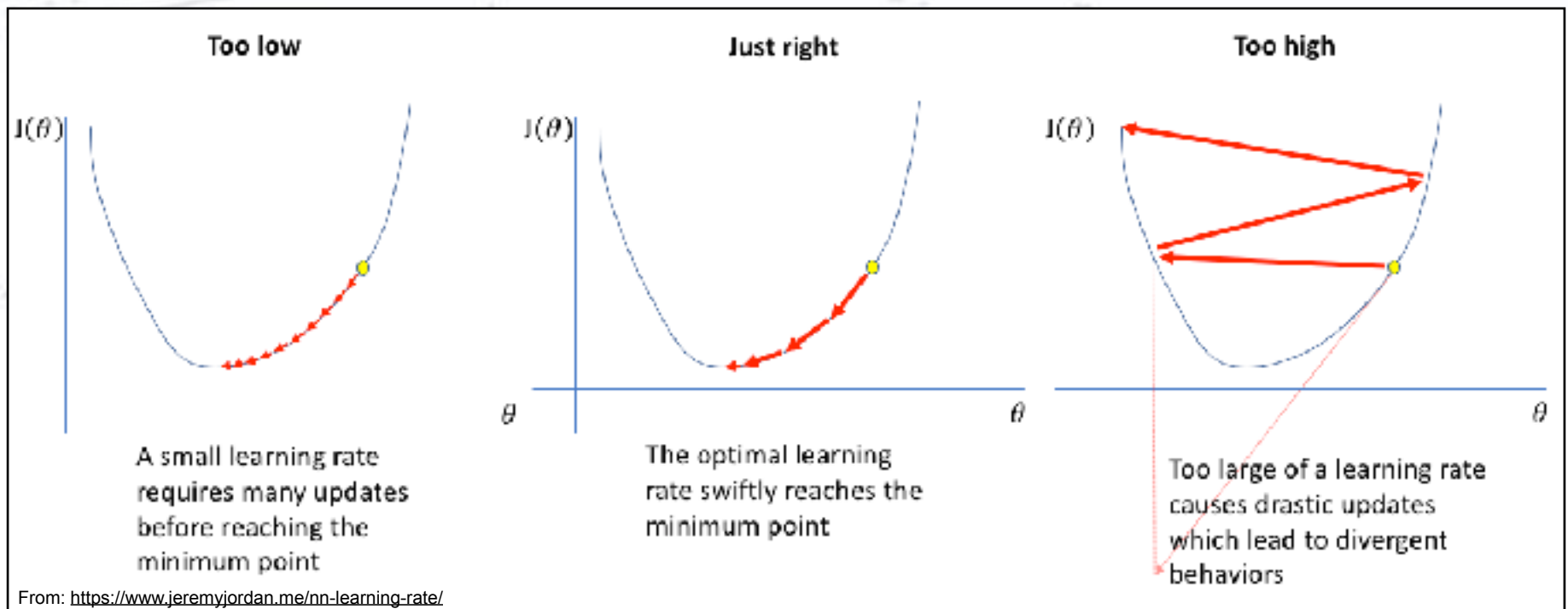
Learning Rate Schedulers

But, there is **no reason to consider a fixed value for the learning rate!**

More practically, one would typically adapt the learning rate to the situation:

- When exploring: Use larger learning rate.
- When exploiting: Use lower learning rate (when converging).

Below is illustrated what happens, when the learning rate is right/wrong.



Choosing Learning Rate

Too low learning rate: Convergence very (too) slow.

Too high learning rate: Random jumps and no convergence.

You want to increase it until it fails and then just below...

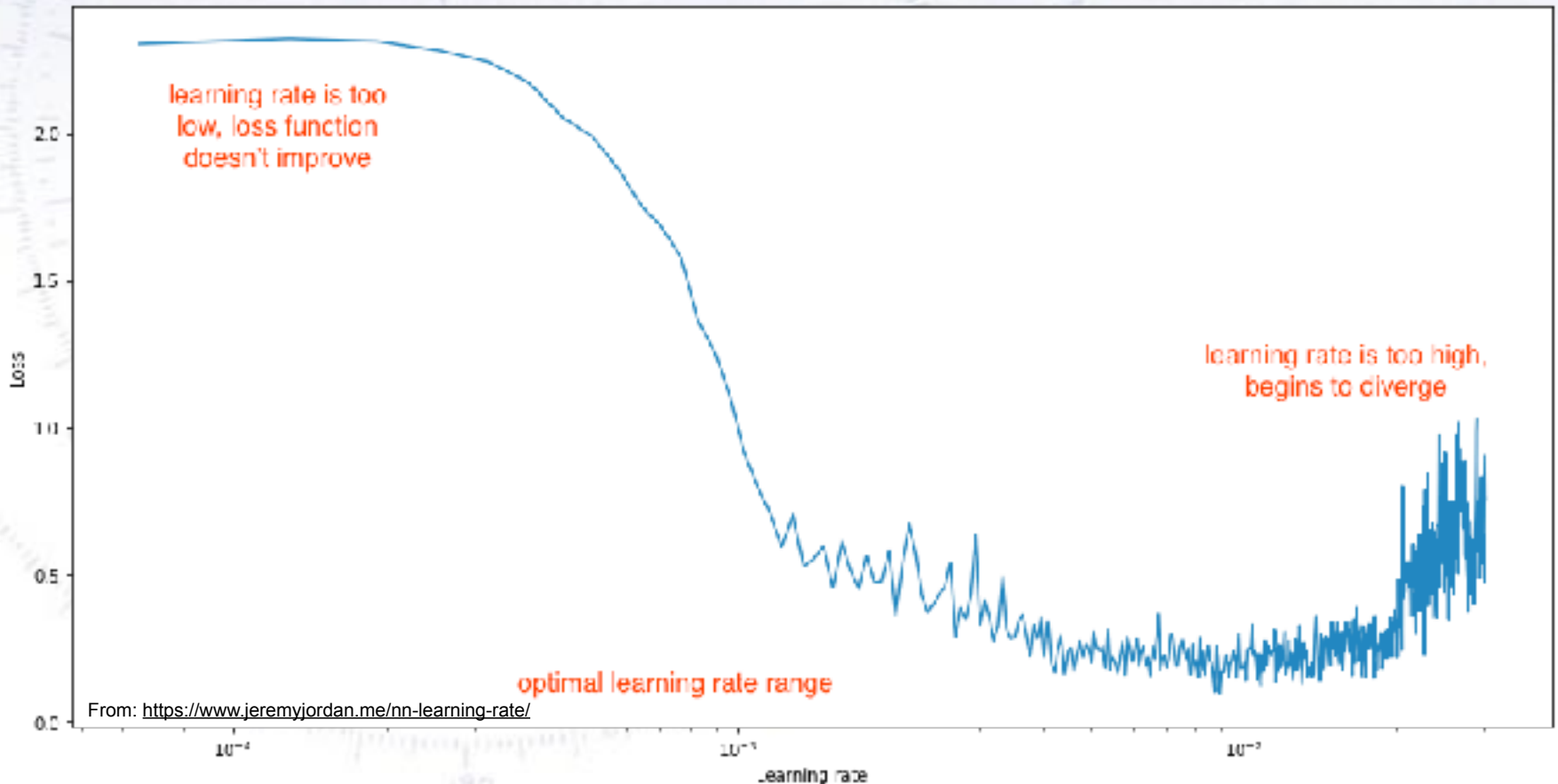


*Ristet brød er let at lave
blot man vil erindre:
når det oser, skal det have
to minutter mindre.*

Piet Hein

Learning Rate Schedulers

First, we want to investigate, which learning rates are relevant (and best) for our problem. **The best learning rate is when the loss decreases the fastest.** Thus we look for the greatest slope of the loss as a function of learning rate:

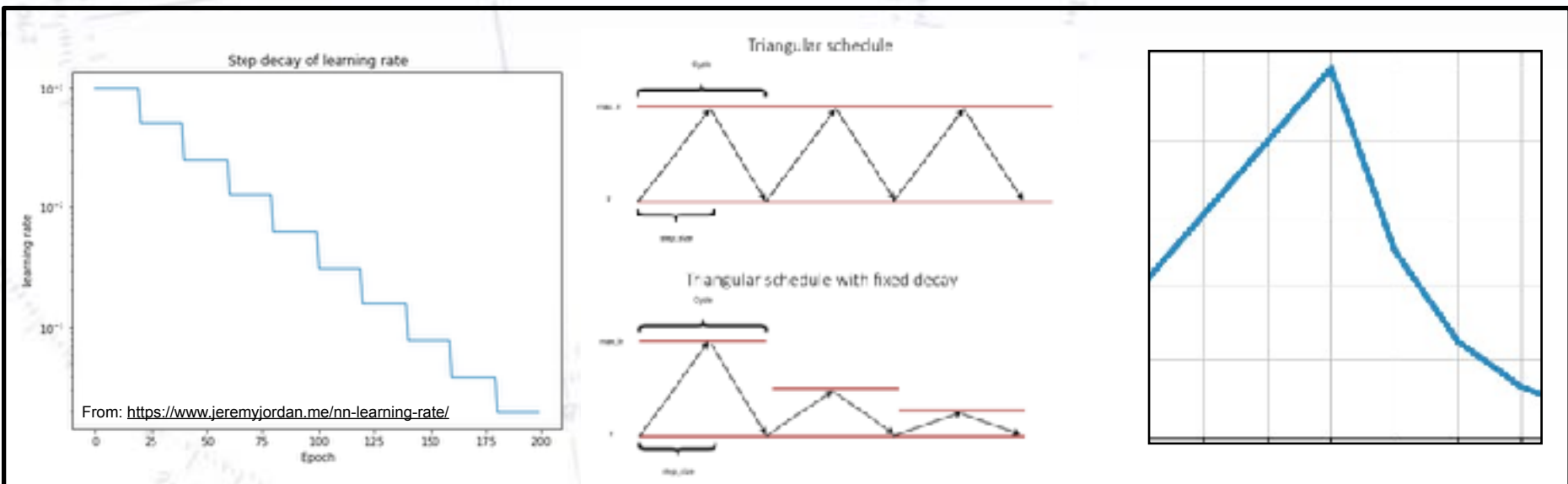


Learning Rate Schedulers

For this reason, Learning Rate Schedulers have been “invented”.

There are MANY different types, and as usual, there is no “right answer”.

However, it is fair to say, that the learning rate is (especially for NNs) the most important Hyper Parameter, and thus it **requires attention**.



Ingredients for ML

So now we know that at least in principle:

- a solution exists (Universal Approximation Theorem) and
- that it can be found (Stochastic Gradient Descent).

But this does not in reality make us capable of getting ML results.

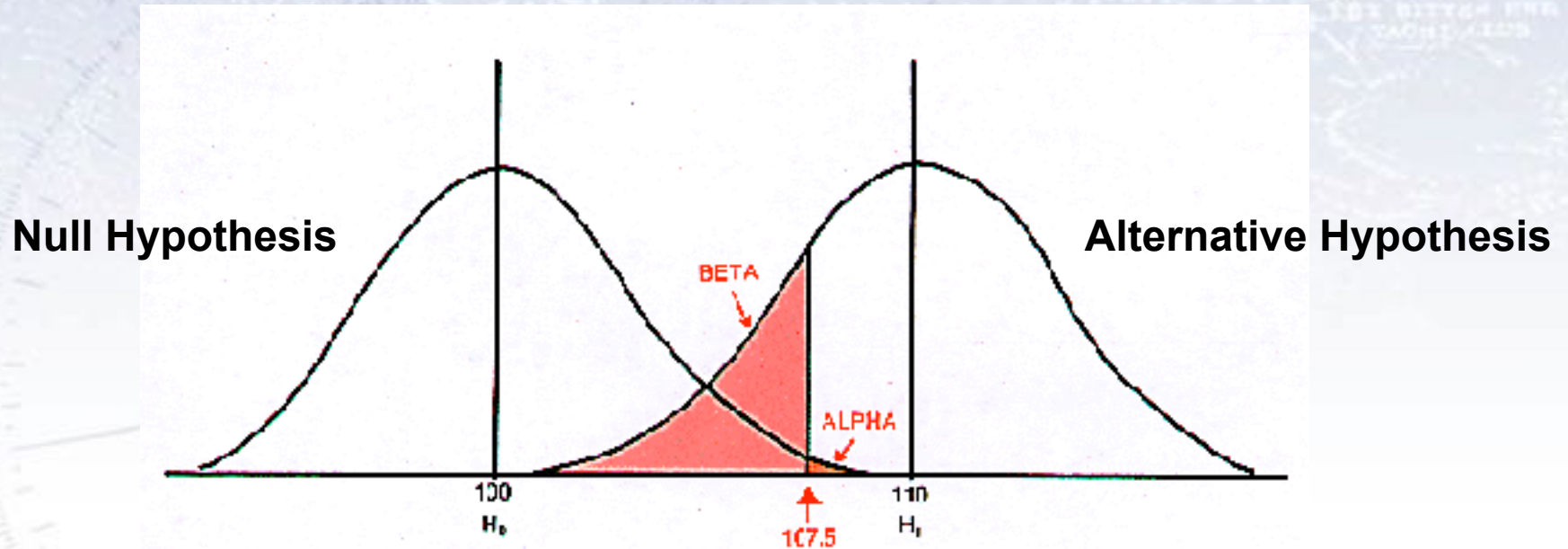
We (at least) also need:

- actual functions/ algorithms for making approximations
Boosted Decision Trees (BDTs) & Neural Networks (NNs)
- knowledge about how to tell them what to learn
Loss functions (and how to minimise these)
- a scheme for how to use the data we have available
Training, validation, and testing samples & Cross Validation

A topographic map of a region, likely in the Pacific Northwest, showing contour lines, rivers, and place names. The map is faded and serves as a background for the text. Visible place names include 'WASHETA' and '1ST BITTCH RNR TACHI / 129'.

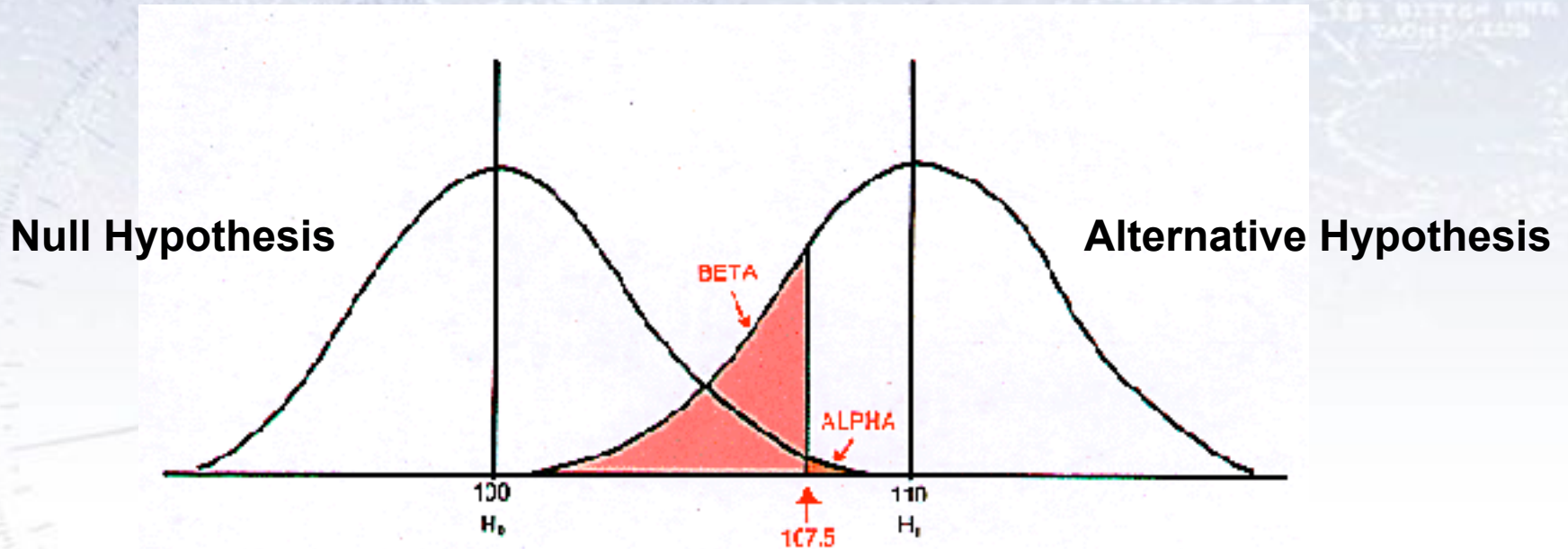
Target of ML

Classification



		REALITY	
		Null is True	Null is False
STATISTICAL DECISION:	Do Not Reject Null	$1 - \alpha$ Correct	β Type II error
	Reject Null	α Type I error	$1 - \beta$ Correct

Classification



Machine Learning typically enables a better separation between hypothesis

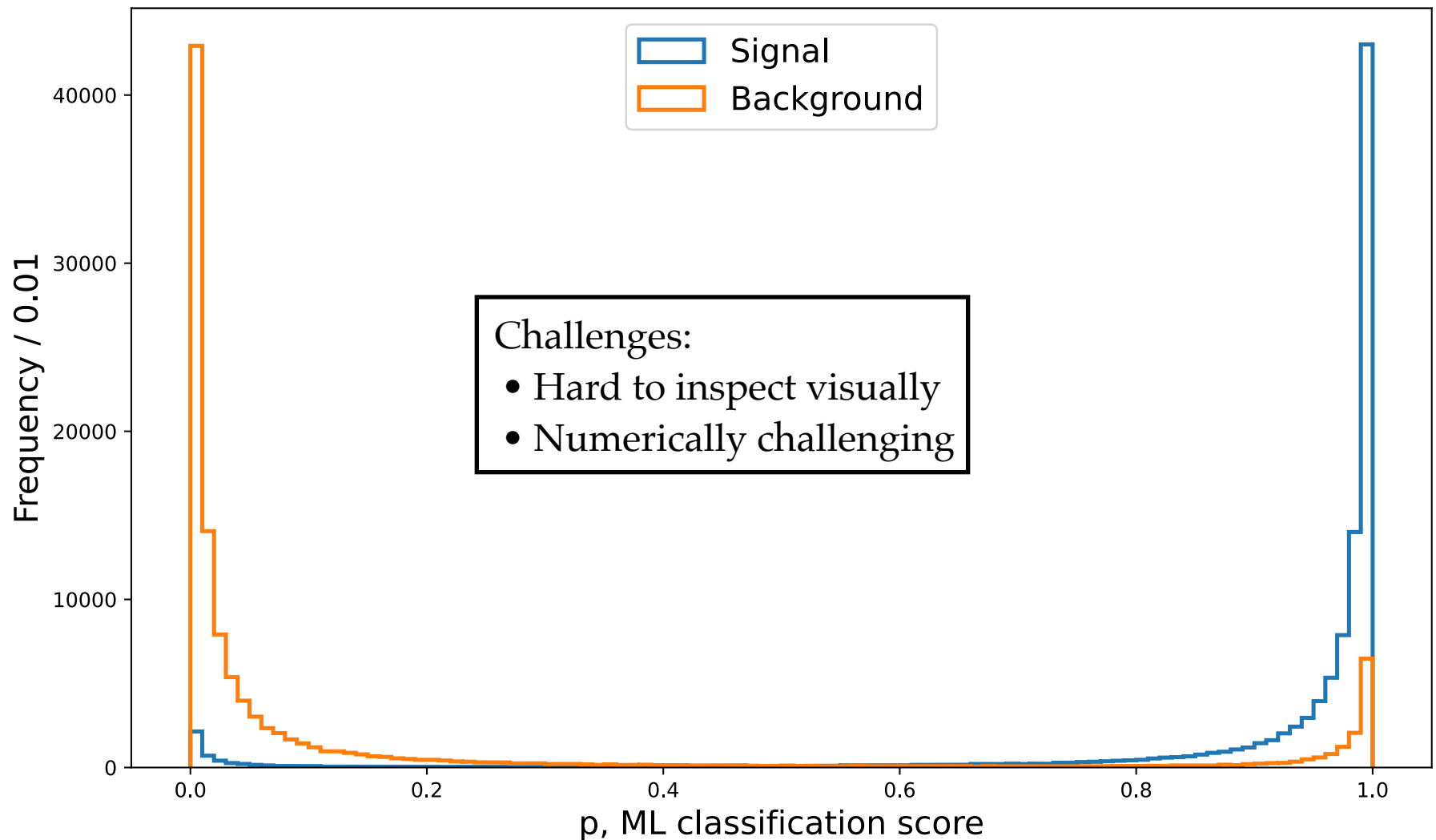
DECISION:

Reject Null

α Type I error	$1 - \beta$ Correct
--------------------------	------------------------

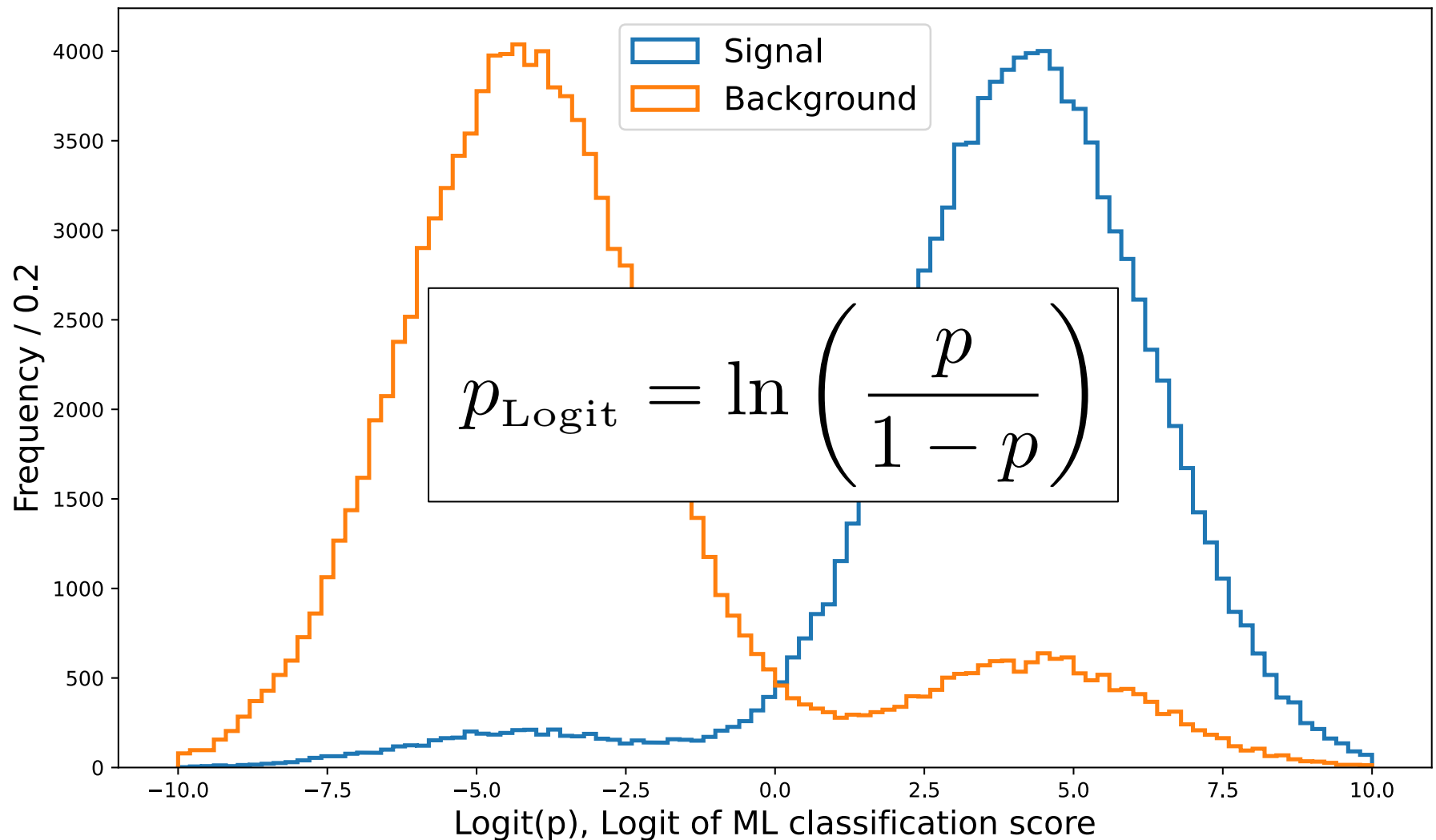
Typical ML Distribution

An ML score distribution from binary classification typically looks as follows:



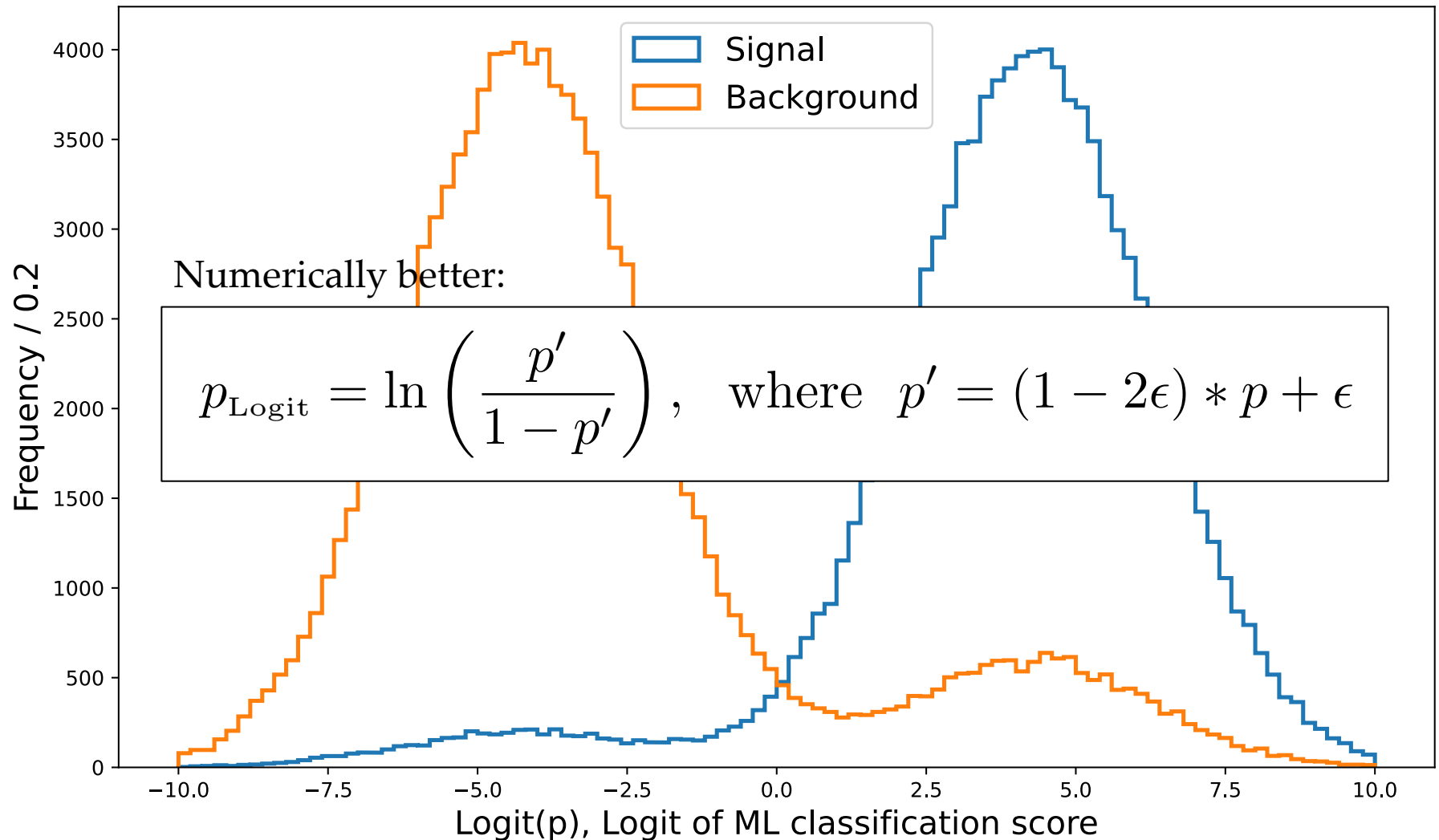
Logit transformation

Once logit transformed, it takes on a “nicer” (and numerically friendly) shape:



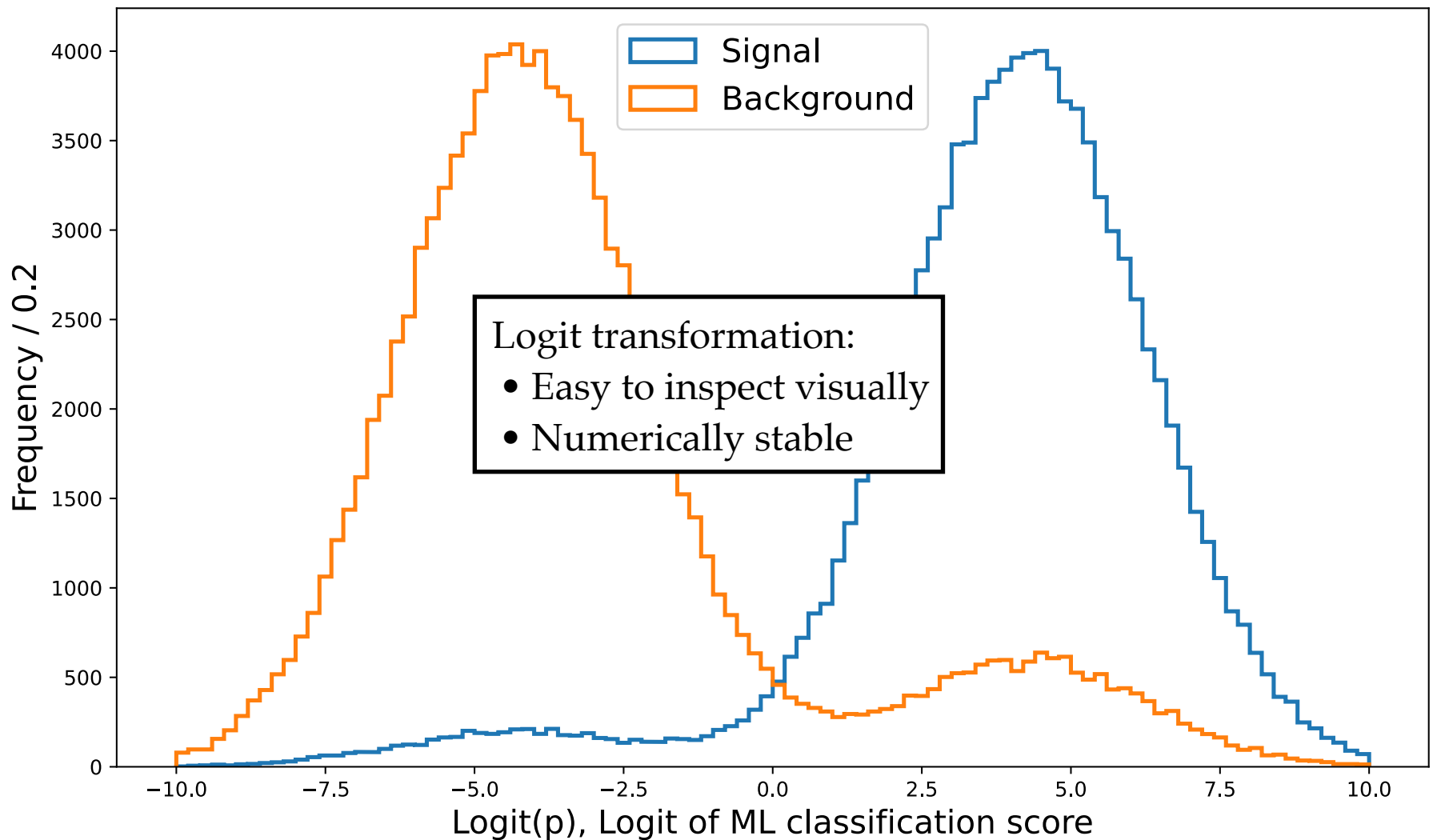
Logit transformation

Once logit transformed, it takes on a “nicer” (and numerically friendly) shape:



Logit transformation

Once logit transformed, it takes on a “nicer” (and numerically friendly) shape:



Hypothesis testing

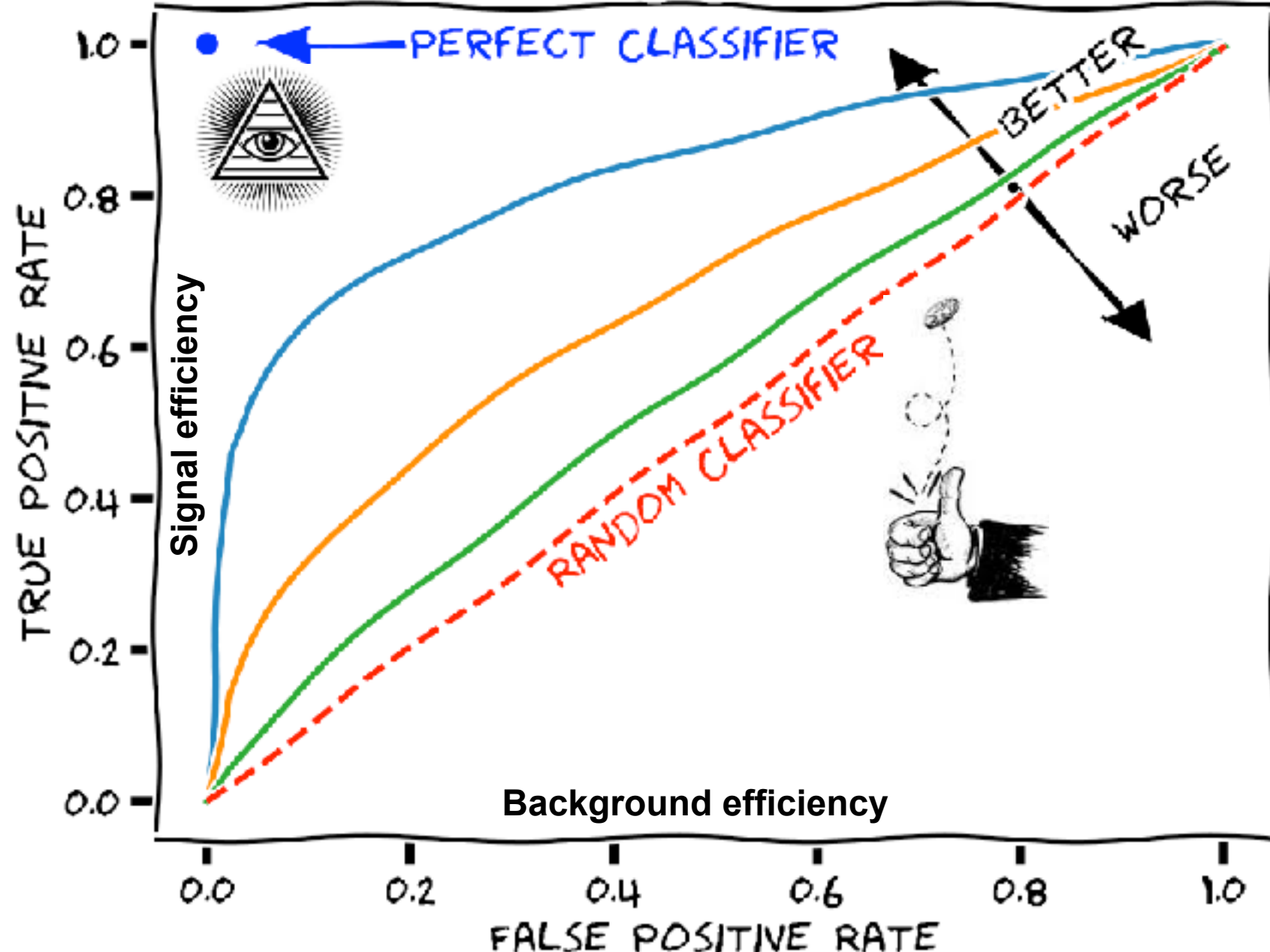
Hypothesis testing is like a criminal trial. The basic “null” hypothesis is **Innocent** (called H_0) and this is the hypothesis we want to test, compared to an “alternative” hypothesis, **Guilty** (called H_1).

Innocence is initially assumed, and this hypothesis is only rejected, if enough evidence proves otherwise, i.e. that the probability of innocence is very small (“beyond reasonable doubt”). This is summarised in a **Contingency Table**:

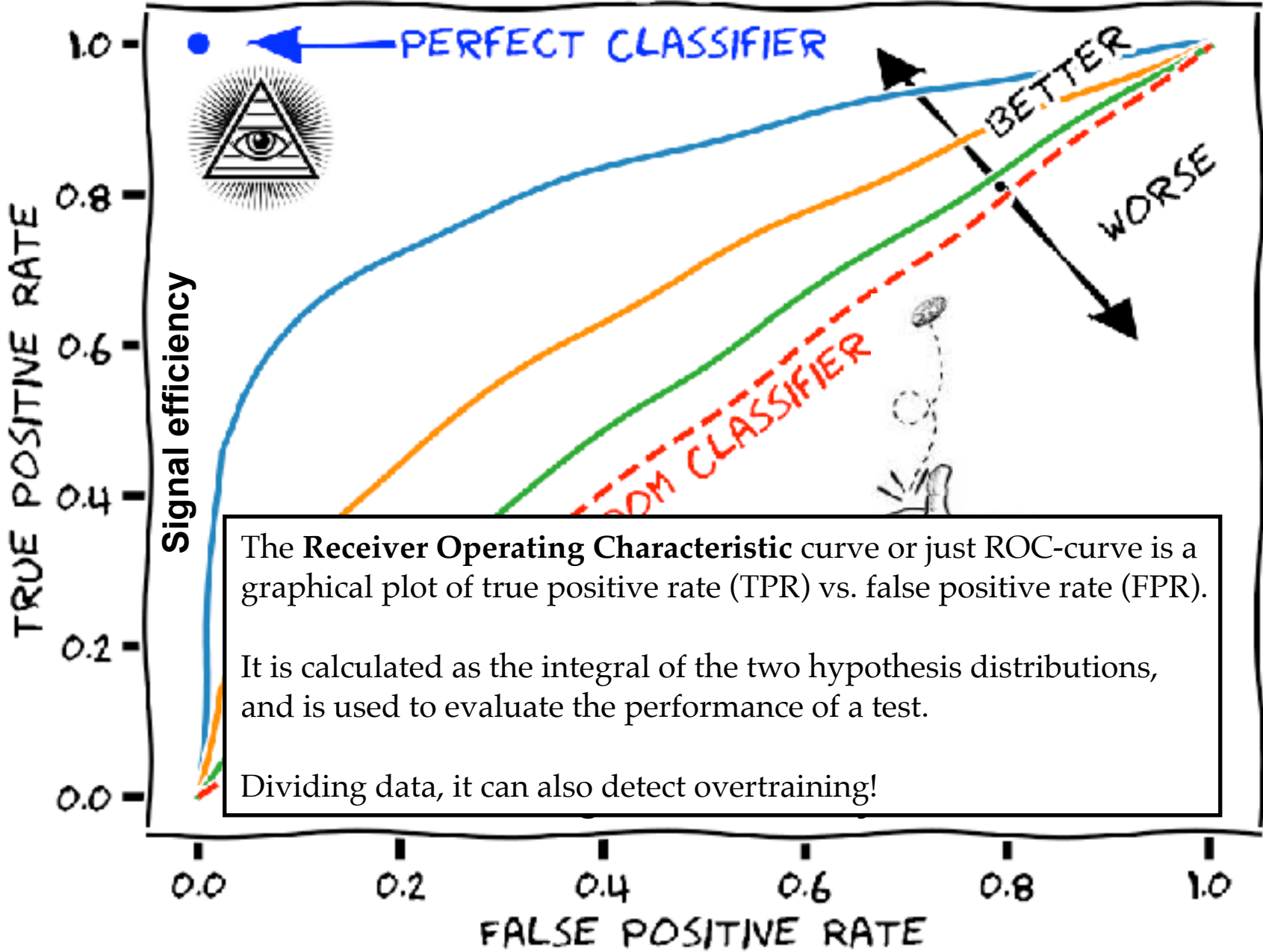
	Truly innocent (H_0 is true)	Truly guilty (H_1 is true)
Acquittal (Accept H_0)	Right decision True Positive (TP)	Wrong decision False Negative (FN)
Conviction (Reject H_0)	Wrong decision False Positive (FP)	Right decision True Negative (TN)

The rate of FP and FN are correlated, and one can only choose one of these!

ROC CURVE



ROC CURVE



The **Receiver Operating Characteristic** curve or just ROC-curve is a graphical plot of true positive rate (TPR) vs. false positive rate (FPR). It is calculated as the integral of the two hypothesis distributions, and is used to evaluate the performance of a test. Dividing data, it can also detect overtraining!

Which metric to use?

There are a ton of metrics in hypothesis testing, see below. However, those in the boxes below are the most central ones.

One metric - not mentioned here - is the Area Under the Curve (AUC), which is simply an integral of the ROC curve (thus 1 is perfect score). This is sometimes used to optimise performance (loss), but not great!

		True condition			
		Condition positive	Condition negative		
Total population				Prevalence = $\frac{\sum \text{Condition positive}}{\sum \text{Total population}}$	Accuracy (ACC) = $\frac{\sum \text{True positive} + \sum \text{True negative}}{\sum \text{Total population}}$
Predicted condition	Predicted condition positive	True positive	False positive, Type I error	Positive predictive value (PPV), Precision = $\frac{\sum \text{True positive}}{\sum \text{Predicted condition positive}}$	False discovery rate (FDR) = $\frac{\sum \text{False positive}}{\sum \text{Predicted condition positive}}$
	Predicted condition negative	False negative, Type II error	True negative	False omission rate (FOR) = $\frac{\sum \text{False negative}}{\sum \text{Predicted condition negative}}$	Negative predictive value (NPV) = $\frac{\sum \text{True negative}}{\sum \text{Predicted condition negative}}$
		True positive rate (TPR), Recall, Sensitivity, probability of detection, Power = $\frac{\sum \text{True positive}}{\sum \text{Condition positive}}$	False positive rate (FPR), Fall-out, probability of false alarm = $\frac{\sum \text{False positive}}{\sum \text{Condition negative}}$	Positive likelihood ratio (LR+) = $\frac{\text{TPR}}{\text{FPR}}$	Diagnostic odds ratio (DOR) = $\frac{\text{LR+}}{\text{LR-}}$
		False negative rate (FNR), Miss rate = $\frac{\sum \text{False negative}}{\sum \text{Condition positive}}$	Specificity (SPC), Selectivity, True negative rate (TNR) = $\frac{\sum \text{True negative}}{\sum \text{Condition negative}}$	Negative likelihood ratio (LR-) = $\frac{\text{FNR}}{\text{TNR}}$	
				F ₁ score = $2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$	

Matthew's Correlation Coefficient

Given a Contingency Table:

	Got well	Remained ill
Medicin	28	5
No Medicin	19	9

One of the commonly used measures of separation the MCC, which (in this case) is the Pearson ρ , and related to the ChiSquare:

$$\text{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

Read more at:

https://en.wikipedia.org/wiki/Phi_coefficient

However, when optimising an algorithm and giving continuous scores in the range $]0,1[$, there are other things to consider (see talk on Loss Functions).

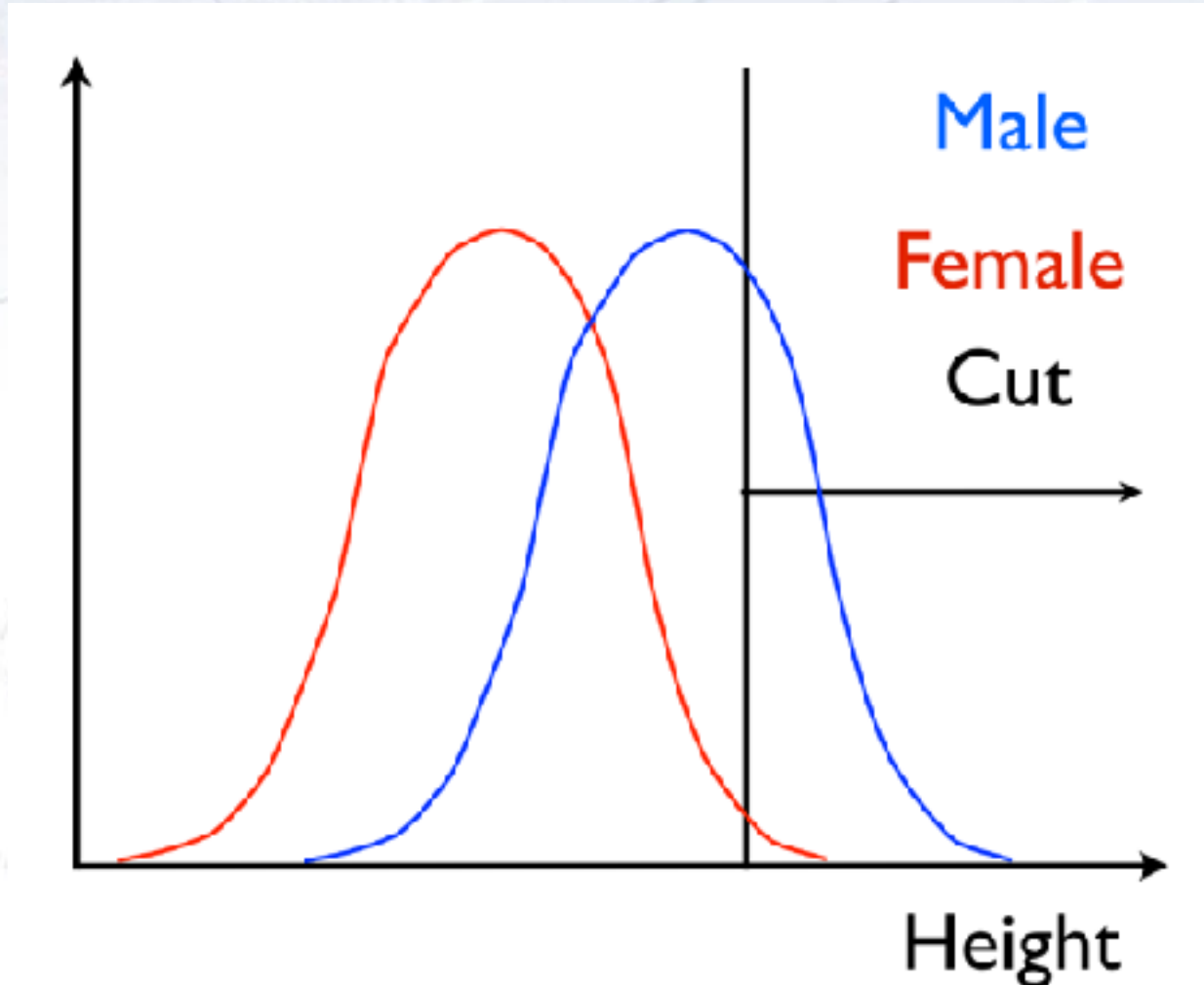


The linear analysis case

Simple Example

Problem: You want to figure out a method for getting sample that is mostly male!

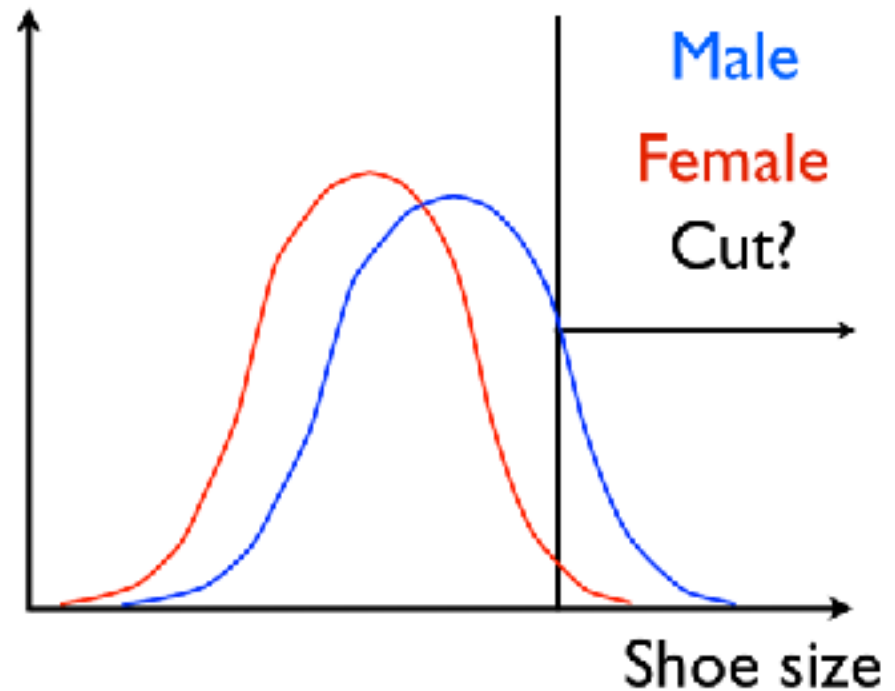
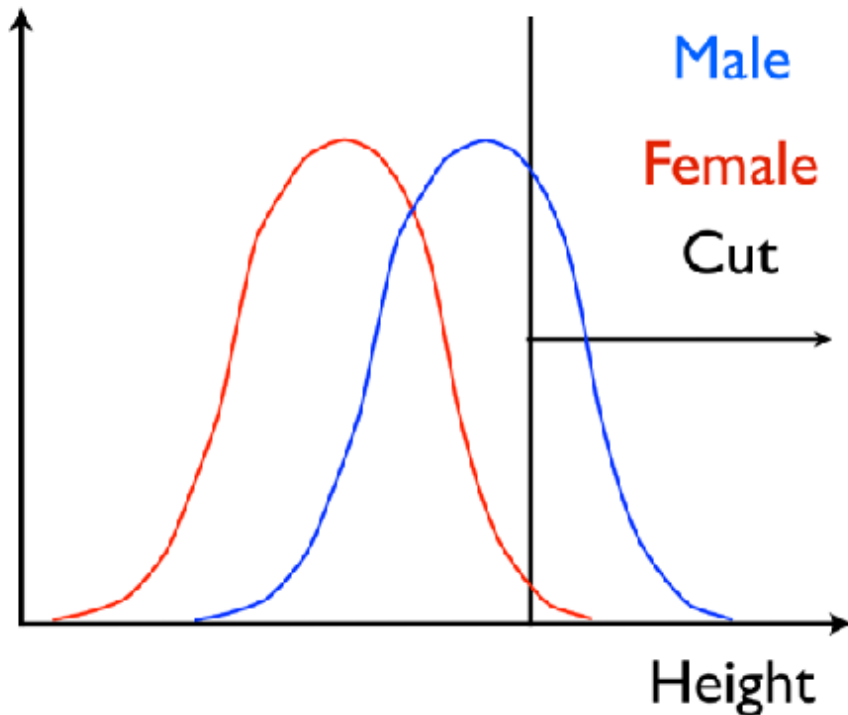
Solution: Gather height data from 10000 people, Estimate cut with 95% purity!



Simple Example

Additional data: The data you find also contains shoe size!

How to use this? Well, it is more information, but should you cut on it?



The question is, what is the best way to use this (possibly correlated) information!

Simple Example

So we look if the data is correlated, and consider the options:

Cut on each var?

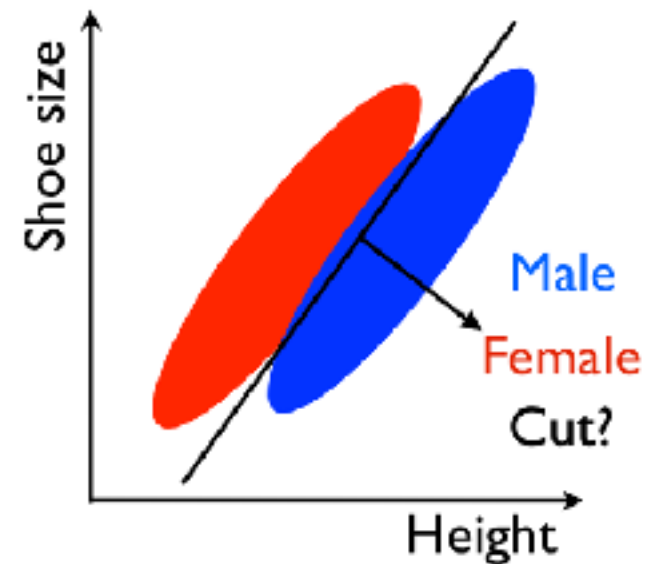
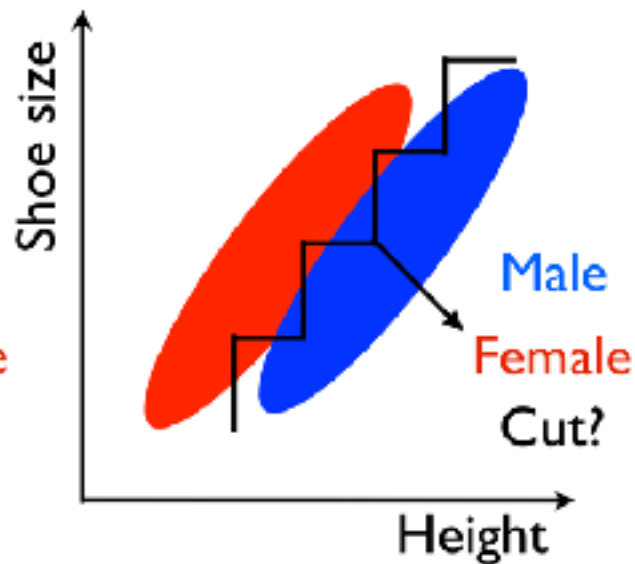
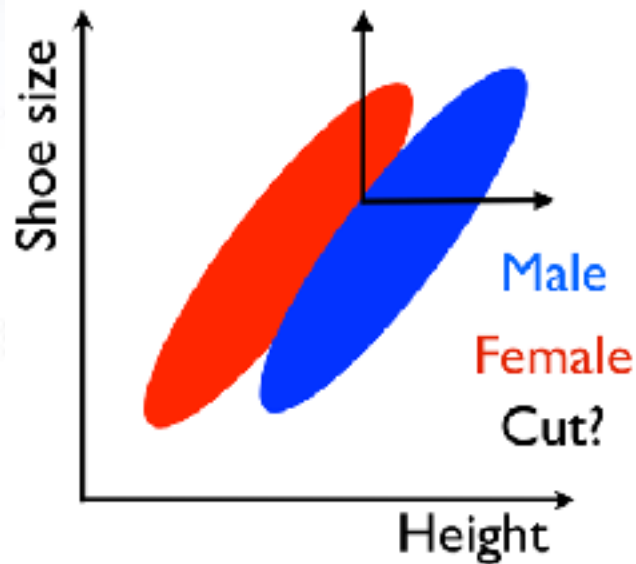
Poor efficiency!

Advanced cut?

**Clumsy and
hard to implement**

Combine var?

**Smart and
promising**



The latter approach is the Fisher discriminant!

It has the advantage of being simple and applicable in many dimensions easily!

Simple Example

So we look if the data is correlated, and consider the options:

Cut on each var?

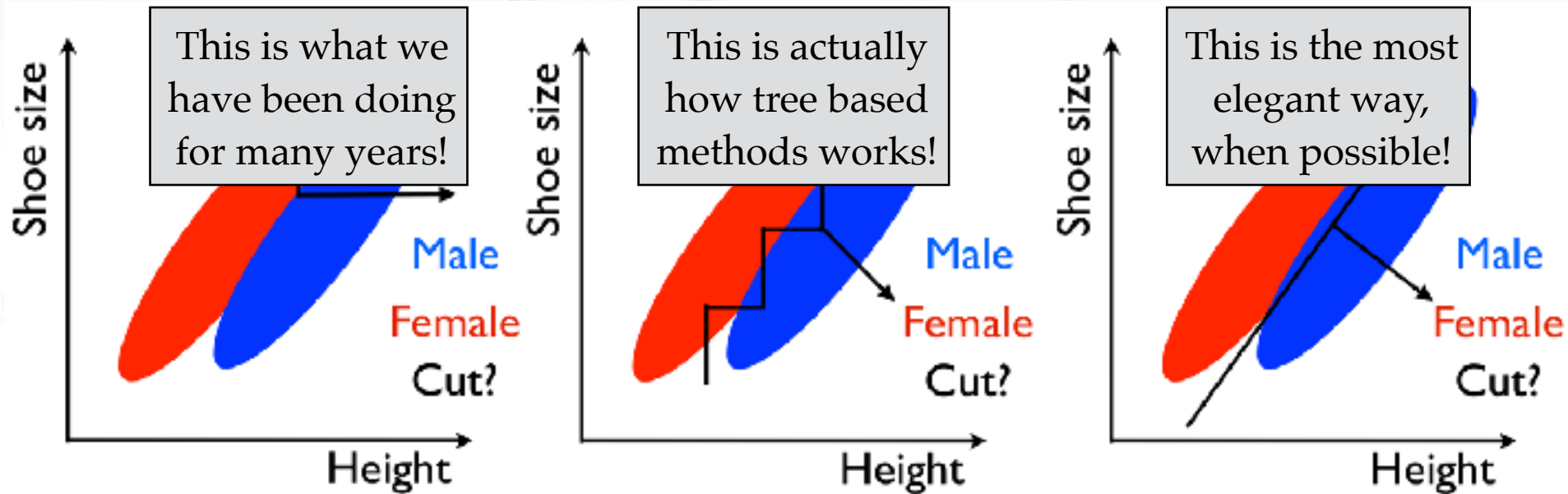
Poor efficiency!

Advanced cut?

**Clumsy and
hard to implement**

Combine var?

**Smart and
promising**



The latter approach is the Fisher discriminant!




It has the advantage of being simple and applicable in many dimensions easily!

Separating data

Fisher's friend, Anderson, came home from picking Irises in the Gaspé peninsula...

180 MULTIPLE MEASUREMENTS IN TAXONOMIC PROBLEMS

Table I

<i>Iris setosa</i>				<i>Iris versicolor</i>				<i>Iris virginica</i>			
Sepal length	Sepal width	Petal length	Petal width	Sepal length	Sepal width	Petal length	Petal width	Sepal length	Sepal width	Petal length	Petal width
5.1	3.5	1.4	0.2	7.0	3.2	4.7	1.4	6.3	3.3	6.0	2.5
4.9	3.0	1.4	0.2	6.4	3.2	4.5	1.5	5.8	2.7	5.1	1.9
4.7	3.2	1.3	0.2	6.9	3.1	4.9	1.5	7.1	3.0	5.9	2.1
4.6	3.1	1.5	0.2	5.5	2.3	4.0	1.3	6.3	2.9	5.6	1.8
											
5.8	4.0	1.2	0.2	5.6	2.9	3.6	1.3	5.8	2.8	5.1	2.4
5.7	4.4	1.5	0.4	6.7	3.1	4.4	1.4	6.4	3.2	5.3	2.3
5.4	3.9	1.3	0.4	5.6	3.0	4.5	1.5	6.5	3.0	5.5	1.8
5.1	3.5	1.4	0.3	5.8	2.7	4.1	1.0	7.7	3.8	6.7	2.2
5.7	3.8	1.7	0.3	6.2	2.2	4.5	1.5	7.7	2.6	6.9	2.3

Fisher's Linear Discriminant

You want to separate two types/classes (A and B) of events using several measurements.

Q: How to combine the variables?

A: Use the Fisher Discriminant:

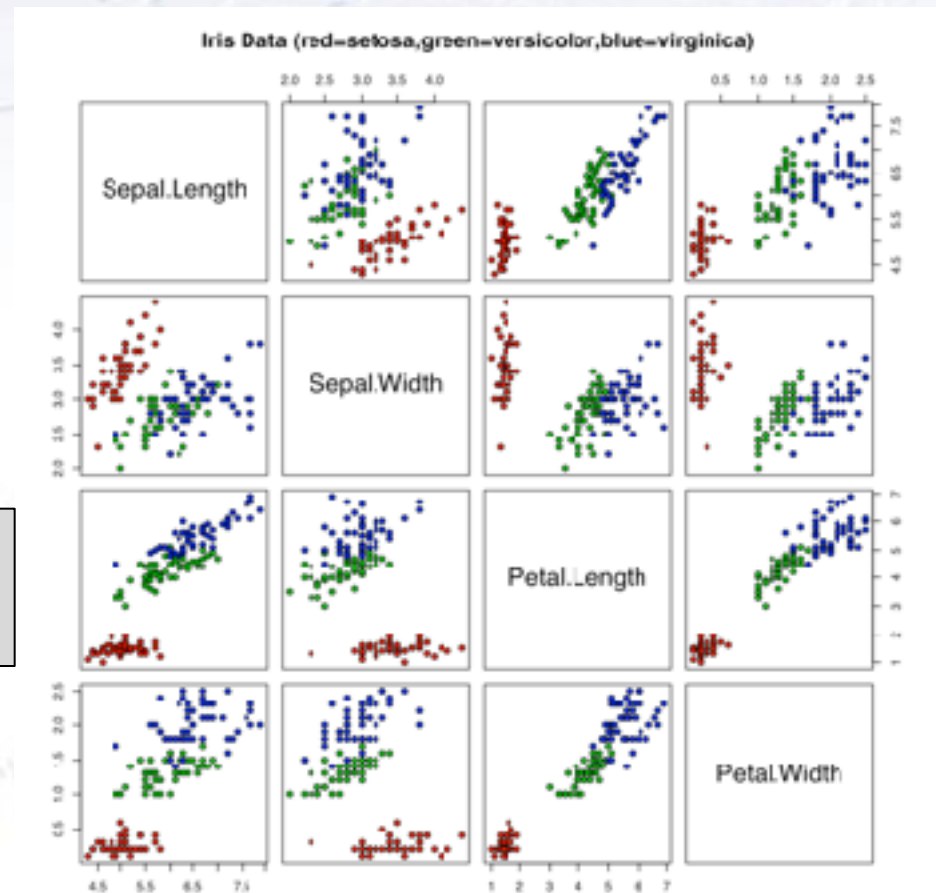
$$\mathcal{F} = w_0 + \vec{w} \cdot \vec{x}$$

Q: How to choose the values of w ?

A: Inverting the covariance matrices:

$$\vec{w} = (\Sigma_A + \Sigma_B)^{-1} (\vec{\mu}_A - \vec{\mu}_B)$$

This can be calculated analytically, and incorporates the linear correlations into the separation capability.



Fisher's Linear Discriminant

The details of the formula are outlined below:

You have two samples, A and B, that you want to separate.

For each input variable (x), you calculate the mean (μ), and form a vector of these.

$$\vec{w} = (\Sigma_A + \Sigma_B)^{-1} (\vec{\mu}_A - \vec{\mu}_B)$$

Using the input variables (x), you calculate the covariance matrix (Σ) for each species (A/B), add these and invert.

Given weights (w), you take your input variables (x) and combine them linearly as follows:

$$\mathcal{F} = w_0 + \vec{w} \cdot \vec{x}$$

F is what you base your decision on.

Fisher's Linear Discriminant

The details of the formula are outlined below:

You have two samples, A and B, that you want

For each input variable (x), you calculate the mean (μ), vector of these.

$$\vec{w} =$$

NOTE:

This is surprisingly close to what Neural Networks do!

They just apply a non-linear function to the result, and repeat!

(A/B), add these and invert.

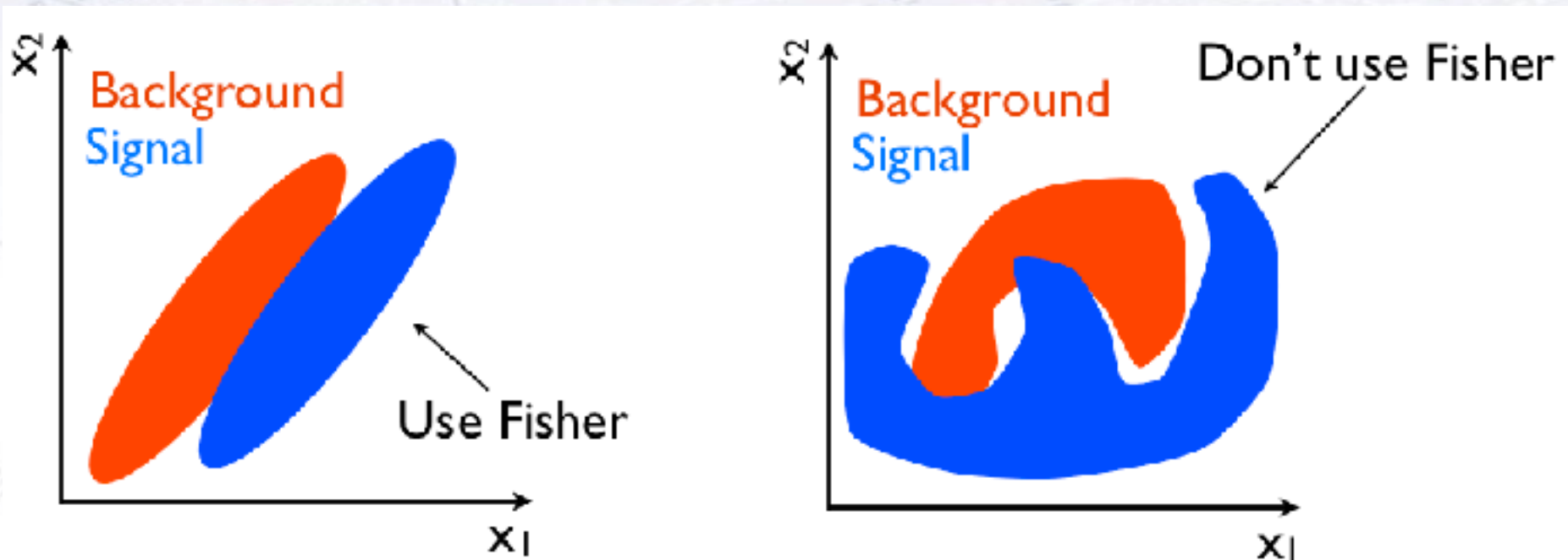
Given weights (w), you take your input variables (x) and combine them linearly as follows:

$$\mathcal{F} = w_0 + \vec{w} \cdot \vec{x}$$

F is what you base your decision on.

Non-linear cases

While the Fisher Discriminant uses all separations and **linear correlations**, it does not perform optimally, when there are **non-linear correlations** present:



If the PDFs of signal and background are known, then one can use a likelihood. But this is **very rarely** the case, and hence one should move on to the Fisher. However, if correlations are non-linear, more “tough” methods are needed...



Tree based models

Decision tree learning

“Tree learning comes closest to meeting the requirements for serving as an off-the-shelf procedure for data mining”,

because it:

- is invariant under scaling and various other transformations of feature values,
- is robust to discontinuous, categorical, and irrelevant features,
- produces inspectable models.

HOWEVER... they are seldom accurate (i.e. most performant)!

[Trevor Hastie, Prof. of Mathematics & Statistics, Stanford]

Again, for tabular data, I tend to disagree with the last statement!

Decision Trees

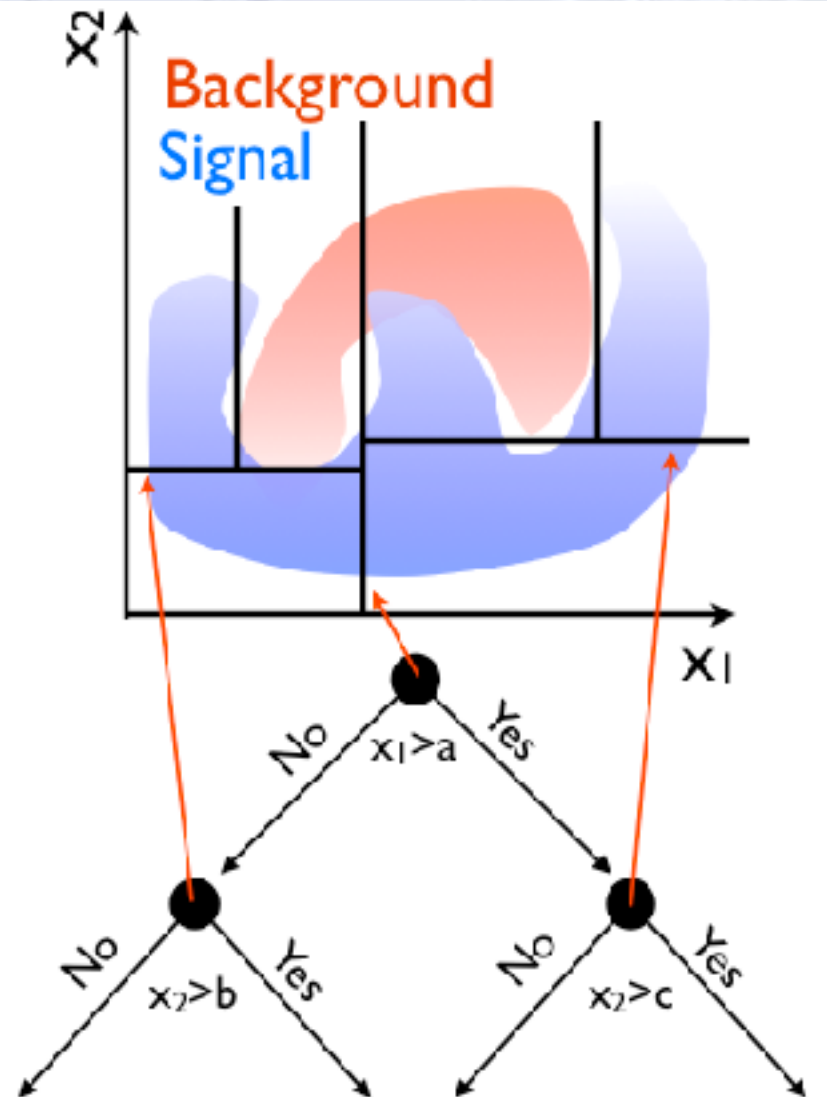
A decision tree divides the parameter space, starting with the maximal separation. In the end each part has a probability of being signal or background.

- Works in 95+% of all problems!
- Fully uses non-linear correlations.

But BDTs require a lot of data for training, and is sensitive to overtraining.

Overtraining can be reduced by limiting the number of nodes and number of trees.

Decision trees are from before 1980!!!



Boosting...

There is no reason, why you can not have more trees. Each tree is a simple classifier, but many can be combined!

To avoid N identical trees, one assigns a higher weight to events that are hard to classify, i.e. boosting:

$$y_{\text{Boost}}(\mathbf{x}) = \frac{1}{N_{\text{collection}}} \cdot \sum_i^{N_{\text{collection}}} \ln(\alpha_i) \cdot h_i(\mathbf{x})$$

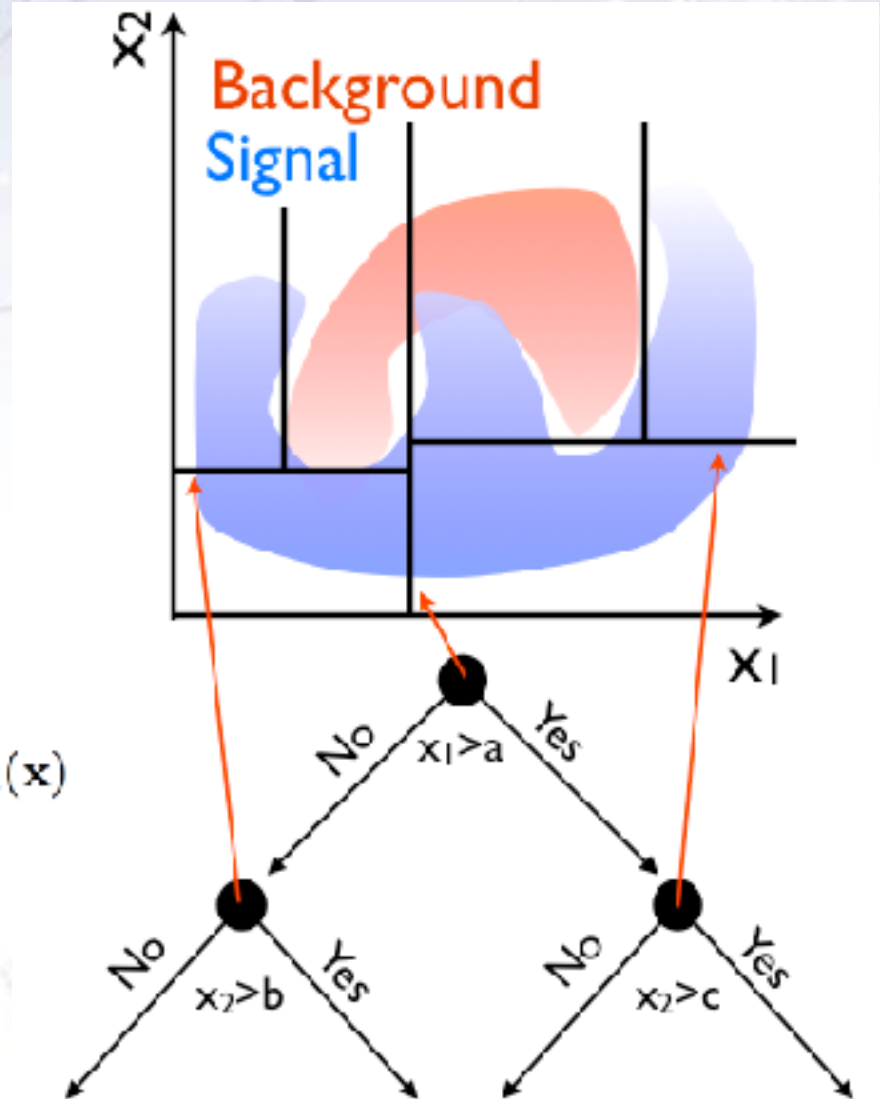
First classifier

Boost weight $\alpha = \frac{1 - \text{err}}{\text{err}}$

Parameters in event N

Individual tree

Boosting is from 1997 (AdaBoost).



Boosting...

There is no reason, why you can not have more trees. Each tree is a simple classifier, but many...

To avoid N identical trees, assign a higher weight to misclassified entries, i.e. boost.

Rerun...
increasing the weight of misclassified entries

First classifier

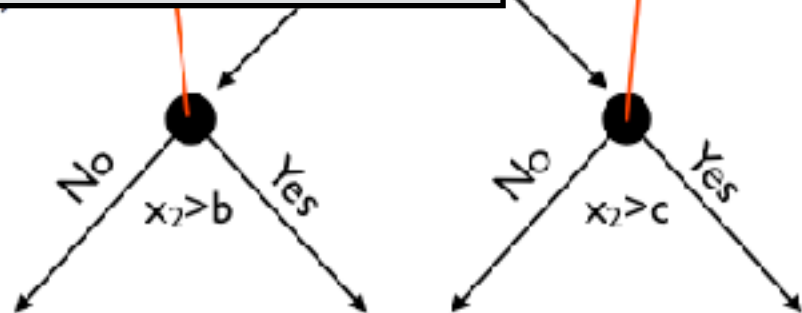
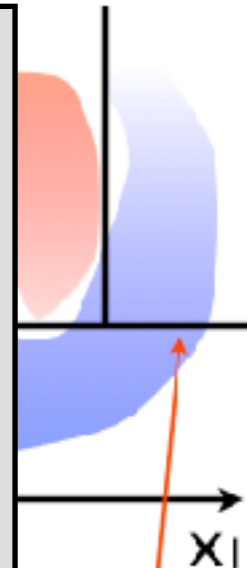
$$y_{\text{Boost}}(\mathbf{x}) = \frac{1}{N_{\text{collection}}} \sum_i$$

Parameters in event N

Individual tree

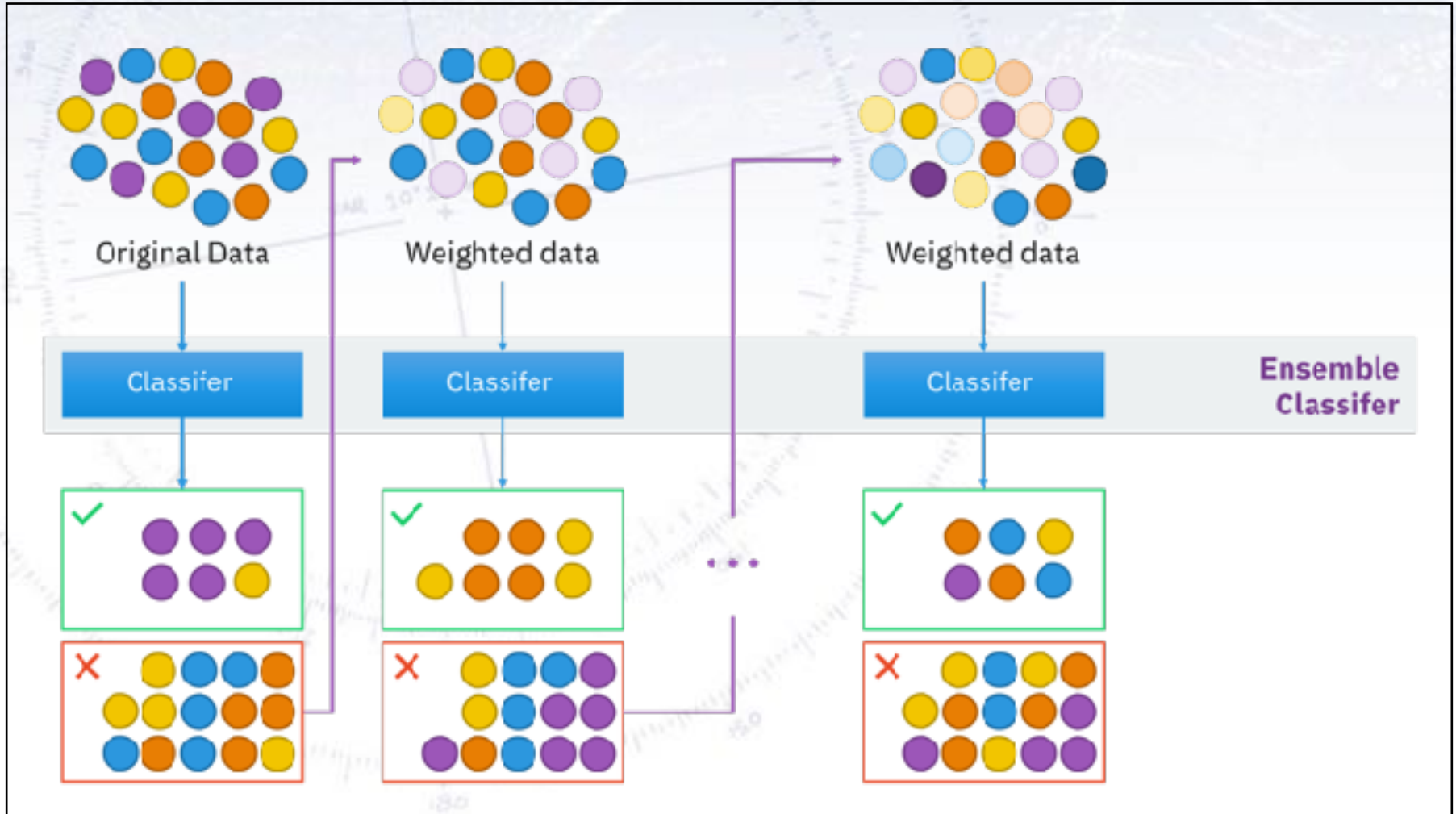
Boosting is from 1997 (AdaBoost).

x_2 ↑
Background



Boosting illustrated

Boosting provides a reweighing scheme giving harder cases higher weights. At the end of training, the trees are collected into an “ensemble classifier”.



Where to split?

How does the algorithm decide which variable to split on and where to split?

There are several ways in which this can be done, and there is a difference between how to do it for classification and regression. But in general, one would like to **make the split, which maximises the improvement gained by doing so.**

In **classification**, one often uses the average binary cross entropy (aka. “log-loss”):

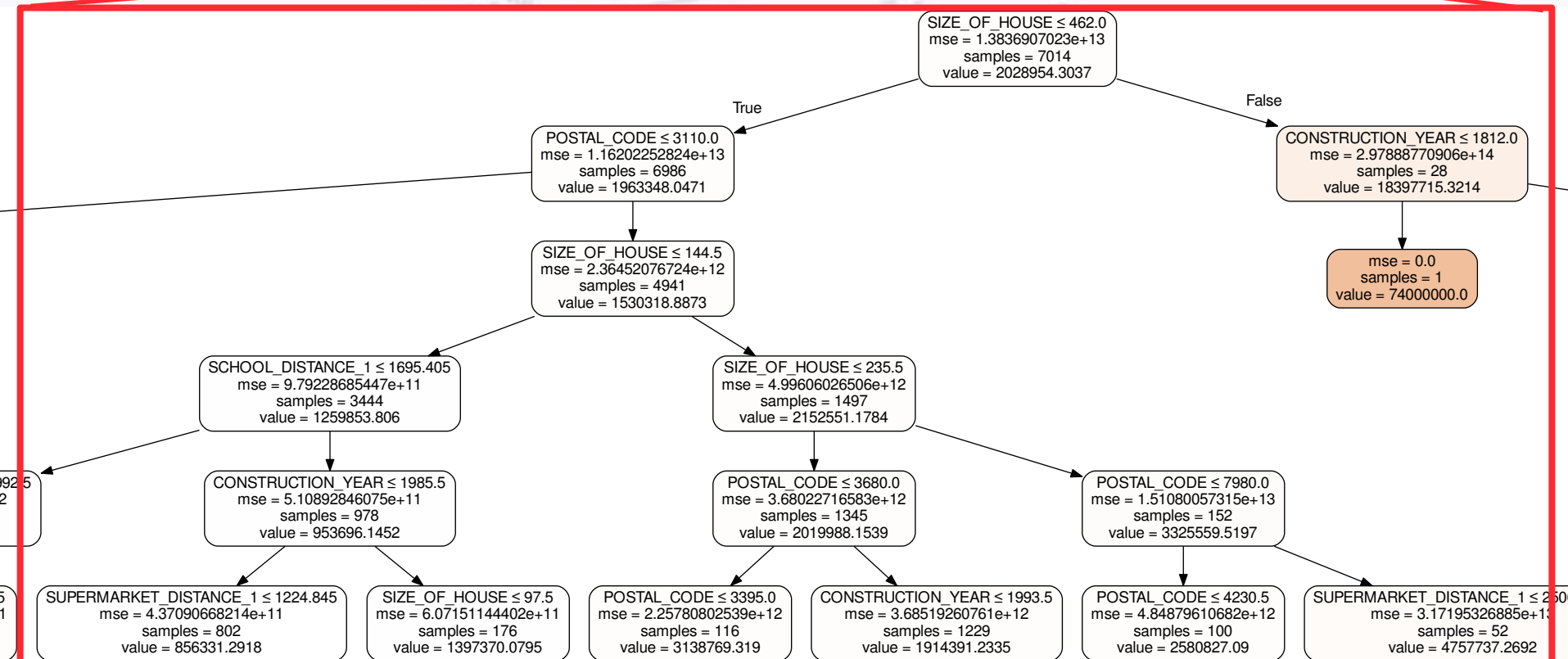
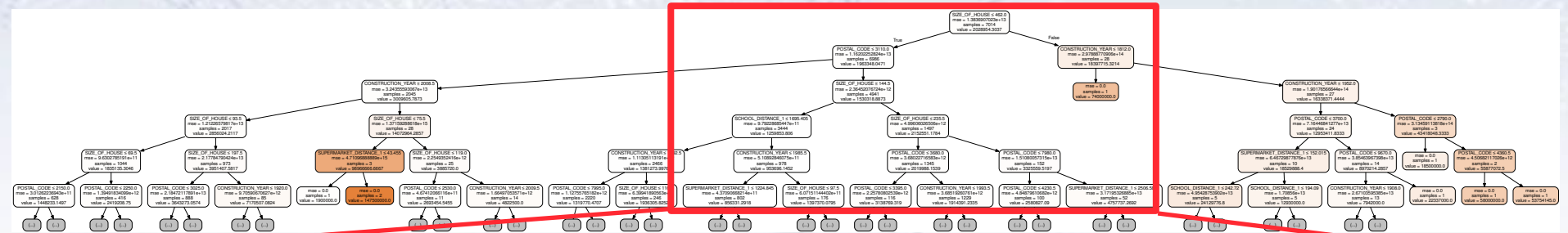
$$\mathcal{L} = -\frac{1}{N} \sum_{n=1}^N [y_n \log \hat{y}_n + (1 - y_n) \log(1 - \hat{y}_n)]$$

Here, y_n is the truth, while \hat{y}_n is the estimate (in $[0,1]$).

Other alternatives include using Gini coefficients, Variance reduction, and even ChiSquare. However, in classification the above is somewhat “standard”.

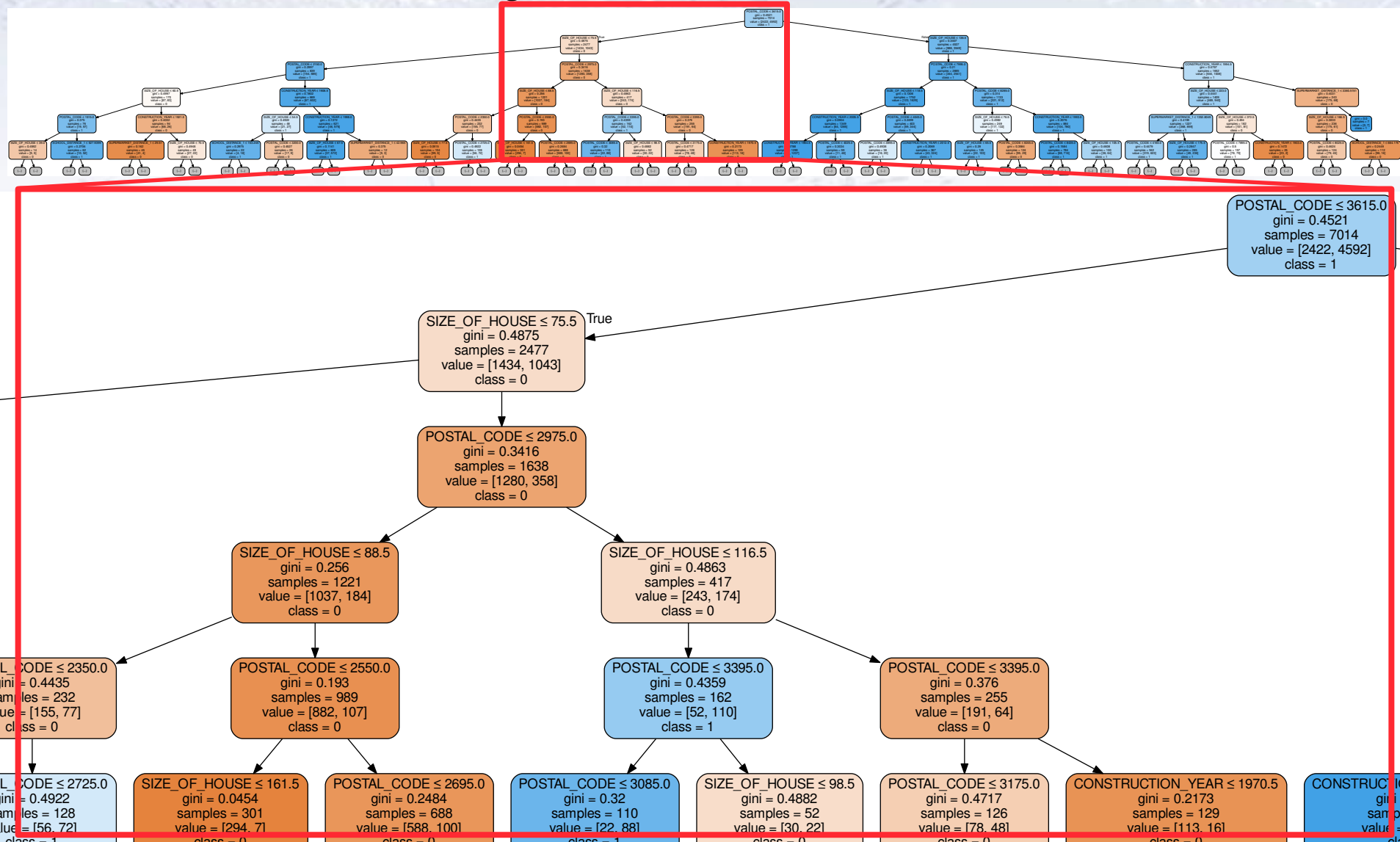
Housing Prices decision tree

Decision tree for estimating the price in the housing prices data set:



Housing Type decision tree

Decision tree for determining, if a house will be sold for more or less than 2Mkr.



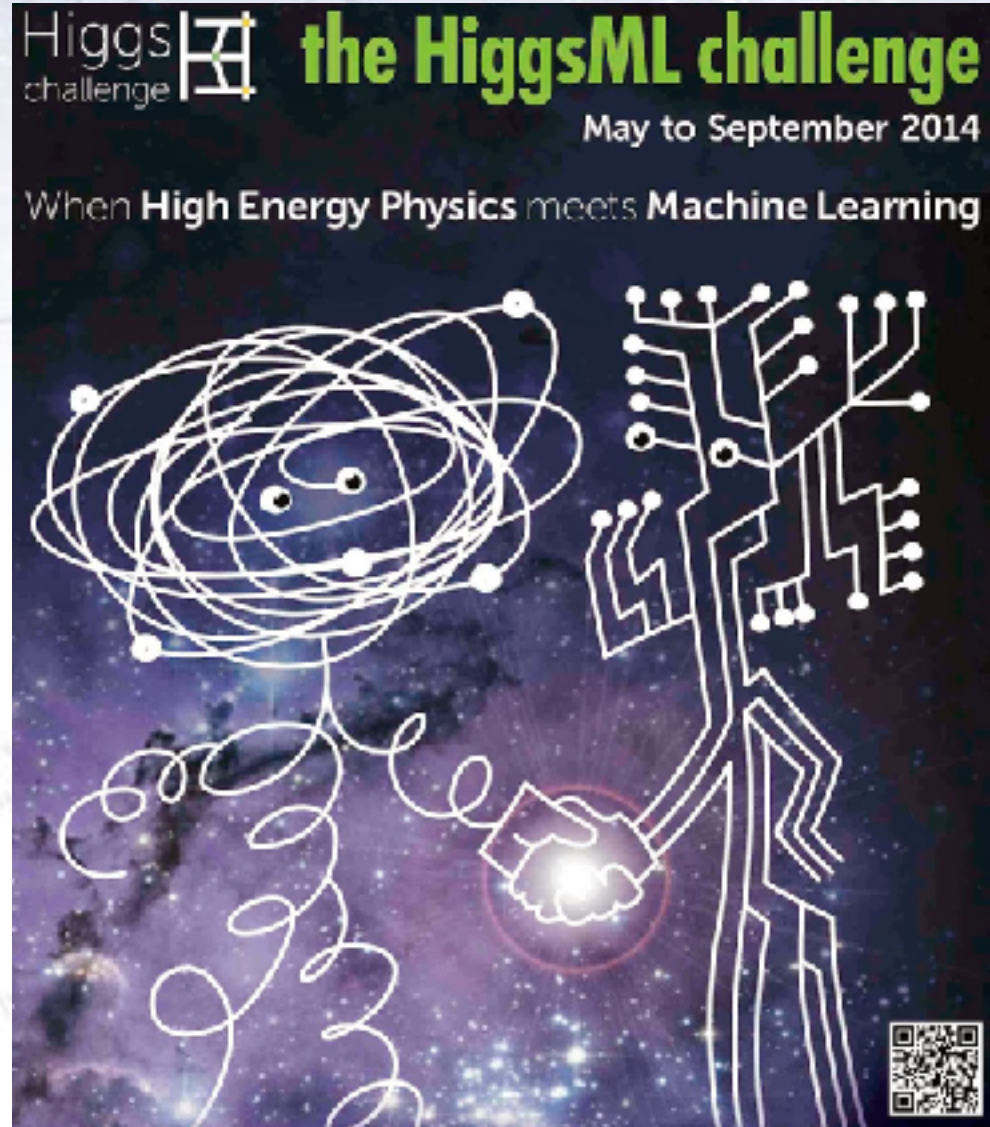


The HiggsML Kaggle Challenge

CERN analyses its data using a vast array of ML methods. CERN is thus part of the community that developpes ML!

After 20 years of using Machine Learning it has now become very widespread (NN, BDT, Random Forest, etc.)

A prime example was the Kaggle “HiggsML Challenge”. Most popular challenge of its time! (1785 teams, 6517 downloads, 35772 solutions, 136 forums)



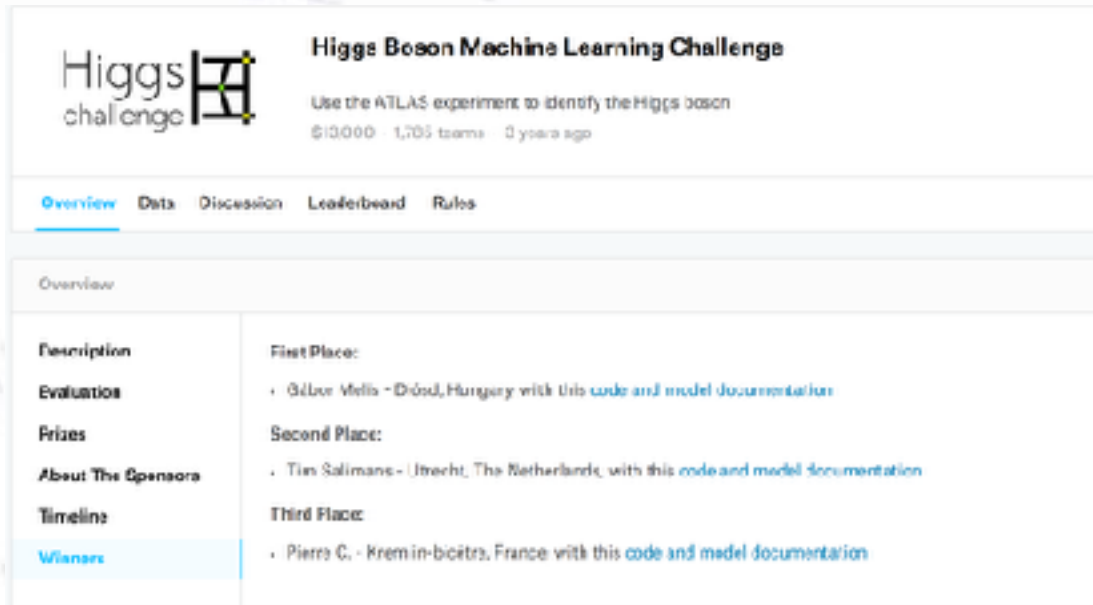
XGBoost history

History [\[edit\]](#)

XGBoost initially started as a research project by Tianqi Chen^[8] as part of the Distributed (Deep) Machine Learning Community (DMLC) group. Initially, it began as a terminal application which could be configured using a libsvm configuration file. After winning the Higgs Machine Learning Challenge, it became well known in the ML competition circles. Soon after, the Python and R packages were built and now it has packages for many other languages like Julia, Scala, Java, etc. This brought the library to more developers and became popular among the [Kaggle](#) community where it has been used for a large number of competitions.^[7]

While Tianqi Chen did not win himself, he provided a method used by about half of the teams, the second place among them!

For this, he got a special award and XGBoost became instantly known in the community.



The screenshot shows the Kaggle page for the Higgs Boson Machine Learning Challenge. The page title is "Higgs Boson Machine Learning Challenge" and the subtitle is "Use the ATLAS experiment to identify the Higgs boson". The page includes a navigation menu with "Overview", "Data", "Discussion", "Leaderboard", and "Rules". The "Overview" section is active and displays the following information:

Description	First Place:
Evaluation	• Gilvor Vello - Dósd, Hungary with this code and model documentation
Prizes	Second Place:
About The Sponsors	• Tin Salimans - Utrecht, The Netherlands, with this code and model documentation
Timeline	Third Place:
Winners	• Pierre C. - Krem-in-boitra, France with this code and model documentation

XGBoost algorithm

The algorithm is documented on the arXiv: 1603.02754

XGBoost: A Scalable Tree Boosting System

Tianqi Chen
University of Washington
tqchen@cs.washington.edu

Carlos Guestrin
University of Washington
guestrin@cs.washington.edu

ABSTRACT

Tree boosting is a highly effective and widely used machine learning method. In this paper, we describe a scalable end-to-end tree boosting system called XGBoost, which is used widely by data scientists to achieve state-of-the-art results on many machine learning challenges. We propose a novel sparsity-aware algorithm for sparse data and weighted quantile sketch for approximate tree learning. More importantly, we provide insights on cache access patterns, data compression and sharding to build a scalable tree boosting system. By combining these insights, XGBoost scales beyond billions of examples using far fewer resources than existing systems.

Keywords

Large-scale Machine Learning

problems. Besides being used as a stand-alone predictor, it is also incorporated into real-world production pipelines for ad click through rate prediction [15]. Finally, it is the de-facto choice of ensemble method and is used in challenges such as the Netflix prize [3].

In this paper, we describe XGBoost, a scalable machine learning system for tree boosting. The system is available as an open source package². The impact of the system has been widely recognized in a number of machine learning and data mining challenges. Take the challenges hosted by the machine learning competition site Kaggle for example. Among the 29 challenge winning solutions³ published at Kaggle's blog during 2015, 17 solutions used XGBoost. Among these solutions, eight solely used XGBoost to train the model, while most others combined XGBoost with neural nets in ensembles. For comparison, the second most popular method, deep neural nets, was used in 11 solutions. The success

XGBoost algorithm

The algorithm is an extension of the decision tree idea (tree boosting), using regression trees with weighted quantiles and being “sparsity aware” (i.e. knowing about lacking entries and low statistics areas of phase space).

Unlike decision trees, each regression tree contains a continuous score on each leaf:

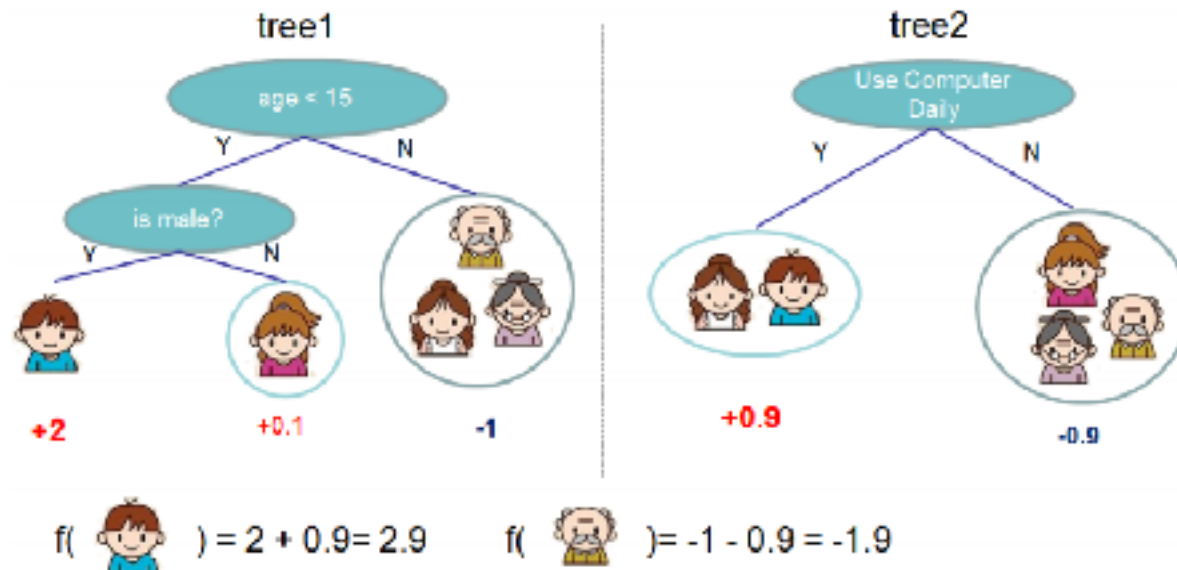
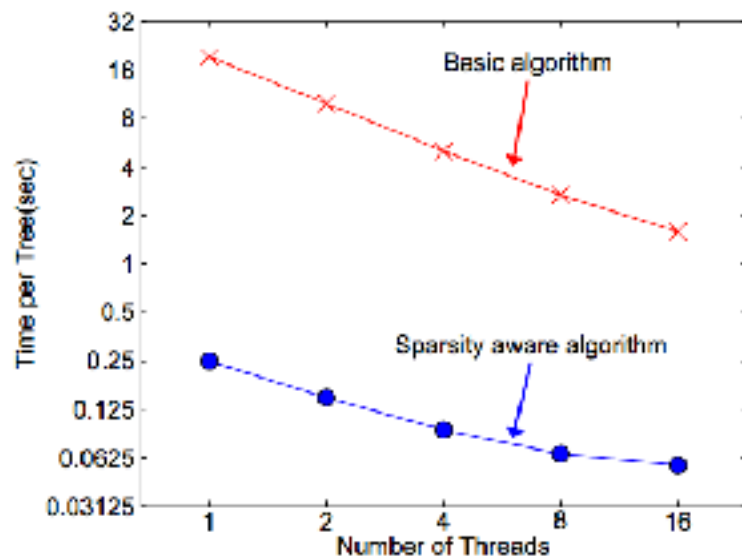


Figure 1: Tree Ensemble Model. The final prediction for a given example is the sum of predictions from each tree.

XGBoost algorithm

The method's speed is partly due to an approximate but fast algorithm to find the best splits.



Algorithm 1: Exact Greedy Algorithm for Split Finding

Input: I , instance set of current node

Input: d , feature dimension

$gain \leftarrow 0$

$G \leftarrow \sum_{i \in I} g_i, H \leftarrow \sum_{i \in I} h_i$

for $k = 1$ **to** m **do**

$G_L \leftarrow 0, H_L \leftarrow 0$

for j in sorted(I , by \mathbf{x}_{jk}) **do**

$G_L \leftarrow G_L + g_j, H_L \leftarrow H_L + h_j$

$G_R \leftarrow G - G_L, H_R \leftarrow H - H_L$

$score \leftarrow \max(score, \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda})$

end

end

Output: Split with max score

Algorithm 2: Approximate Algorithm for Split Finding

for $k = 1$ **to** m **do**

 Propose $S_k = \{s_{k1}, s_{k2}, \dots, s_{kl}\}$ by percentiles on feature k .

 Proposal can be done per tree (global), or per split (local).

end

for $k = 1$ **to** m **do**

$G_{kv} \leftarrow \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} g_j$

$H_{kv} \leftarrow \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} h_j$

end

Follow same step as in previous section to find max score only among proposed splits.

XGBoost algorithm

In order to “punish” complexity, the cost-function has a regularised term also:

$$\mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

$$\text{where } \Omega(f) = \gamma T + \frac{1}{2} \lambda \|w\|^2$$

Table 1: Comparison of major tree boosting systems.

System	exact greedy	approximate global	approximate local	out-of-core	sparsity aware	parallel
XGBoost	yes	yes	yes	yes	yes	yes
pGBRT	no	no	yes	no	no	yes
Spark MLlib	no	yes	no	no	partially	yes
H2O	no	yes	no	no	partially	yes
scikit-learn	yes	no	no	no	no	no
R GBM	yes	no	no	no	partially	no

XGBoost

As it turns out, XGBoost is not only very performant but also very fast...

The most important factor behind the success of XGBoost is its scalability in all scenarios. The system runs more than ten times faster than existing popular solutions on a single machine and scales to billions of examples in distributed or memory-limited settings. The scalability of XGBoost is due to several important systems and algorithmic optimizations.

But this will of course only last for so long - new algorithms see the light of day every week... day?

— — — — — — — — — — shortly after — — — — — — — — — —

Meanwhile, LightGBM has seen the light of day, and it is even faster...

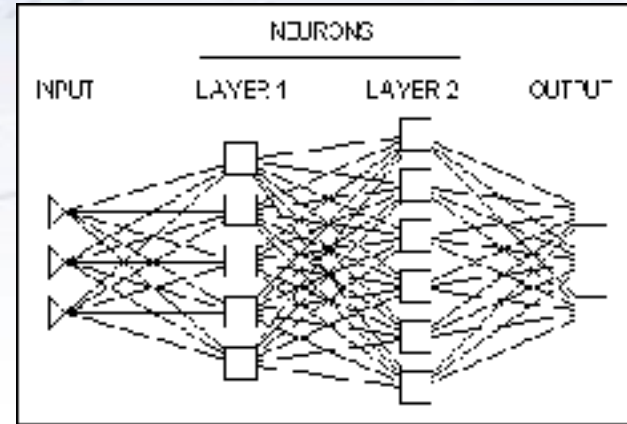
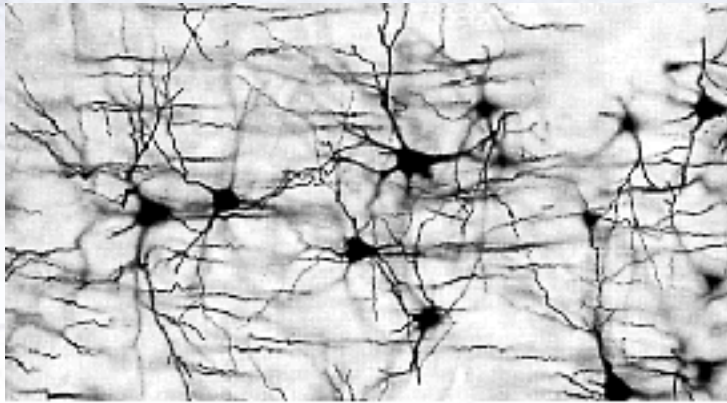
Which algorithm takes the crown: Light GBM vs XGBOOST?

Very good blog with introduction to tree based learning



Neural Network models

Neural Networks (NN)



*In machine learning and related fields, artificial neural networks (ANNs) are computational models inspired by an animal's central nervous systems (in particular the brain) which is capable of **machine learning** as well as **pattern recognition**.*

*Neural networks have been used to solve a wide variety of tasks that are hard to solve using ordinary rule-based programming, including **computer vision** and **speech recognition**.*

[Wikipedia, Introduction to Artificial Neural Network]

A “Linear Network”

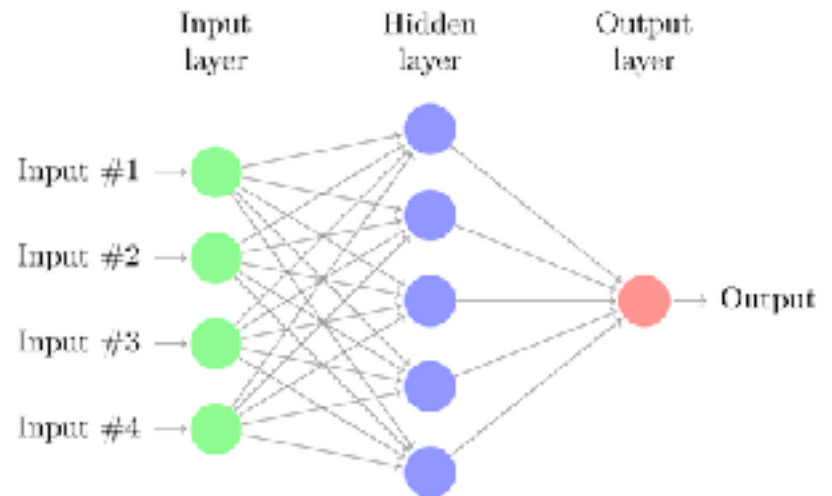
Imagine that we consider a “Linear Network”, and use the (simplest) architecture:
A single layer (linear) perceptron:

$$t(x) = a_0 + \sum a_i x_i$$

As can be seen, this is simply a **linear regression in multiple dimensions** or the (linear) Fisher Discriminant.

Well, then we could consider putting in a hidden (linear) layer:

$$tt(x) = t(a_0 + \sum a_i x_i)$$

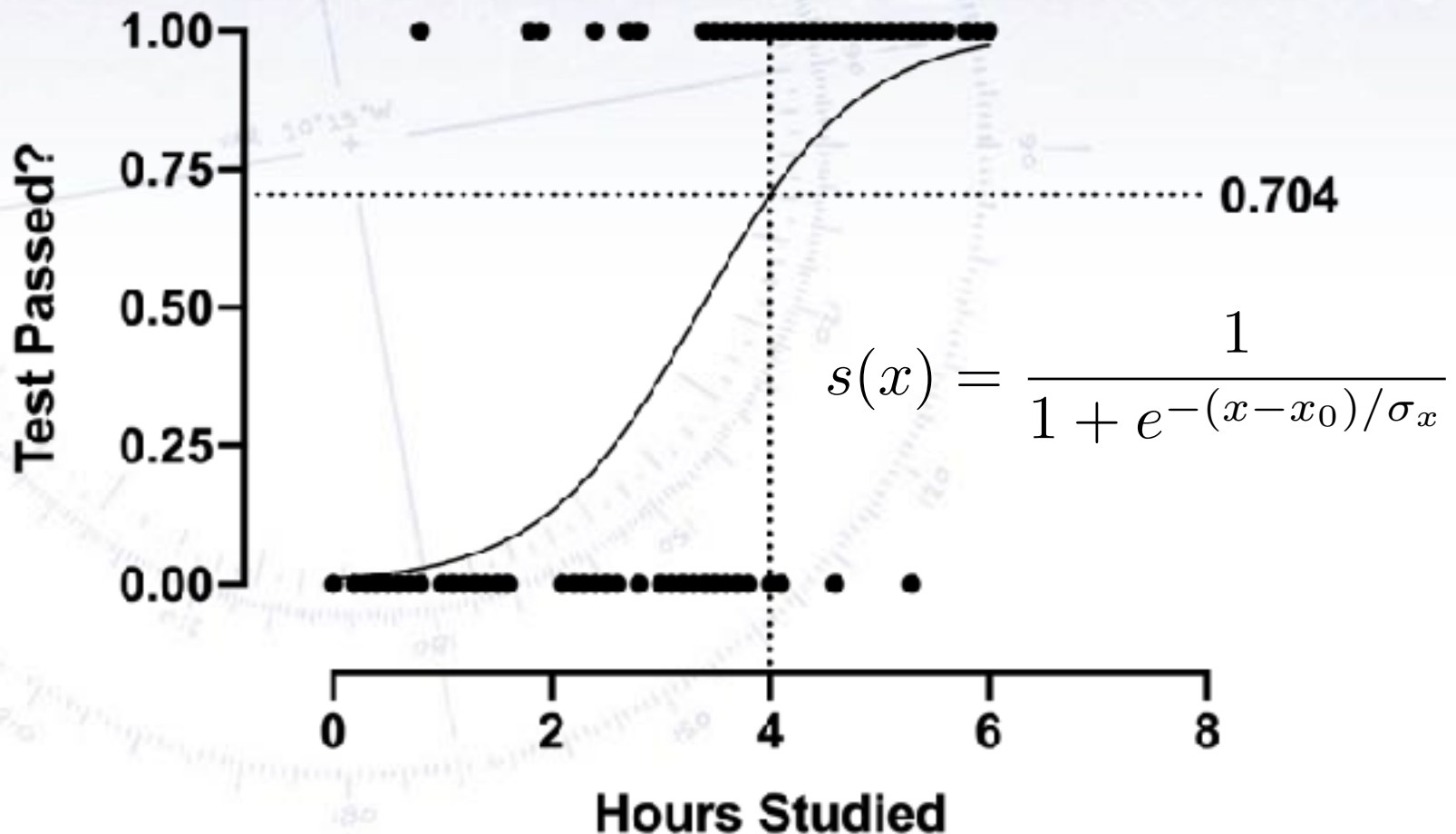


However, this doesn't help anything
as combination of linear functions remain linear. It boils down to the Fisher again!

What we need is something non-linear in the function...

Logistic Regression

Though the word “regression” suggests otherwise, this is in fact a way of doing classification, as the “regression” is usually for a score (s) in the interval [0,1].

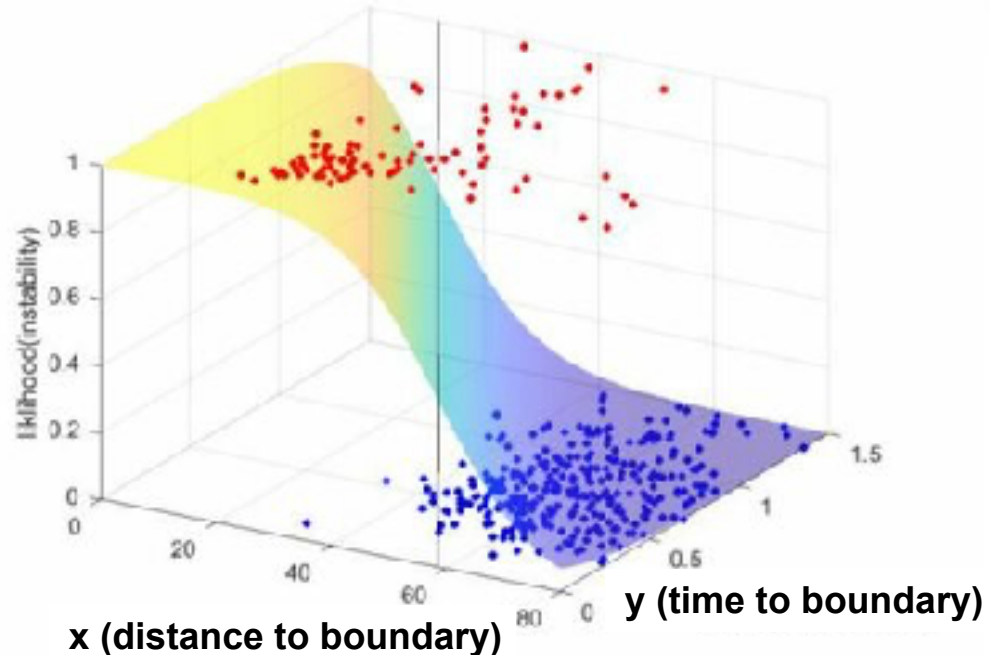
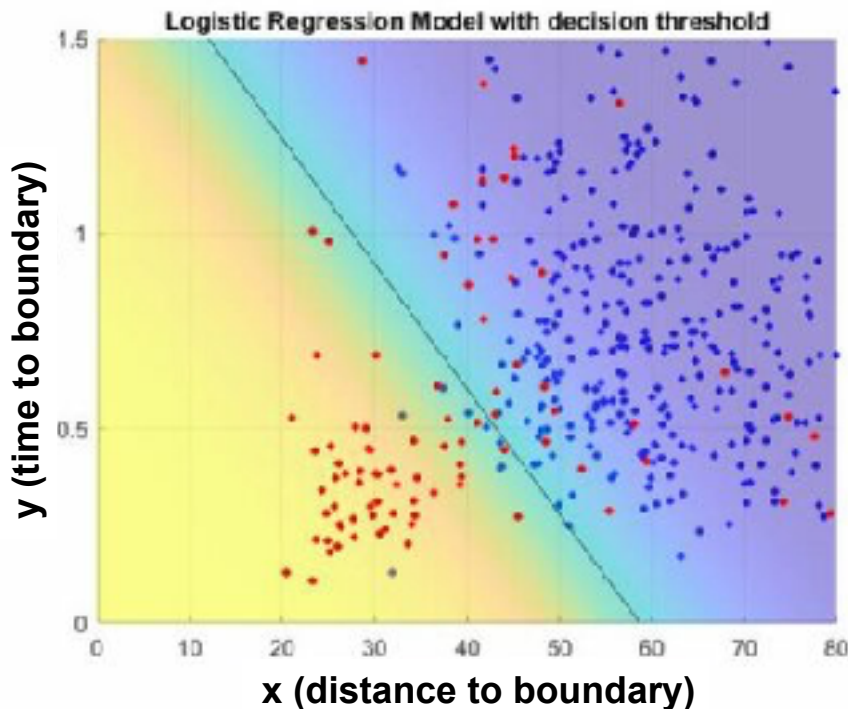


Logistic Regression

Though the word “regression” suggests otherwise, this is in fact a way of doing classification, as the “regression” is usually for a score (s) in the interval [0,1].

The model expands naturally with more parameters:

$$s(x) = \frac{1}{1 + e^{-\frac{(x-x_0)/\sigma_x - (y-y_0)/\sigma_y}{\sigma_x^2 + \sigma_y^2}}}$$



Neural Networks

Neural Networks combine the input variables using a “activation” function $s(x)$ to assign, if the variable indicates signal or background.

The simplest is a single layer perceptron:

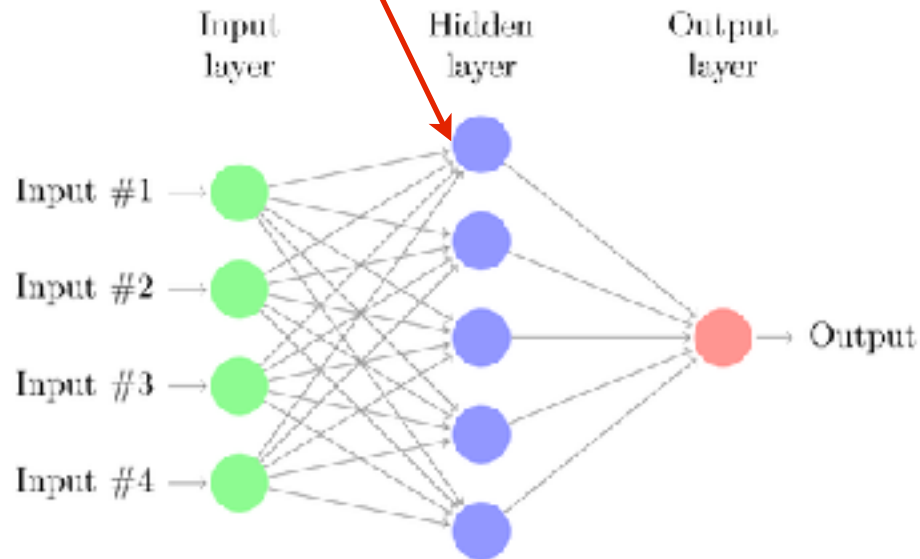
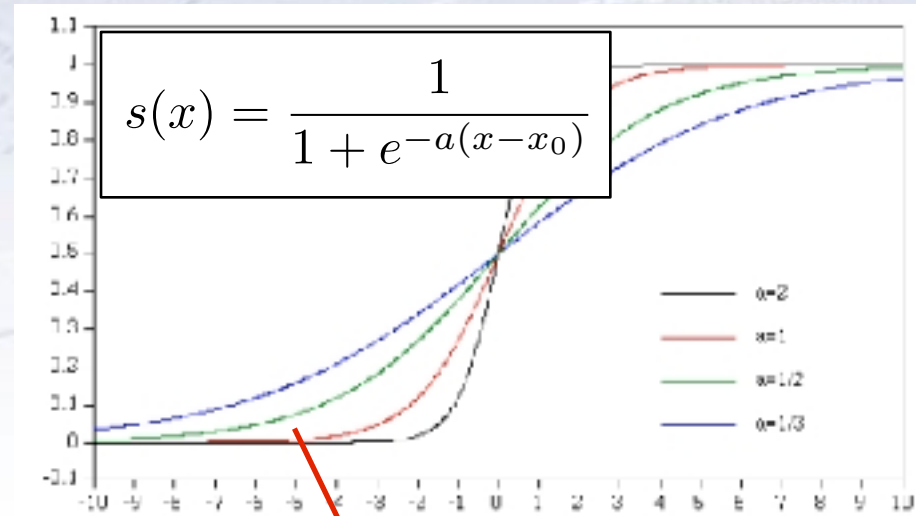
$$t(x) = s \left(a_0 + \sum a_i x_i \right)$$

This can be generalised to a multilayer perceptron (shown right, 1 hidden layer):

$$t(x) = s \left(a_i + \sum a_i h_i(x) \right)$$

$$h_i(x) = s \left(w_{i0} + \sum w_{ij} x_j \right)$$

Activation function can be any “sigmoidal” function.

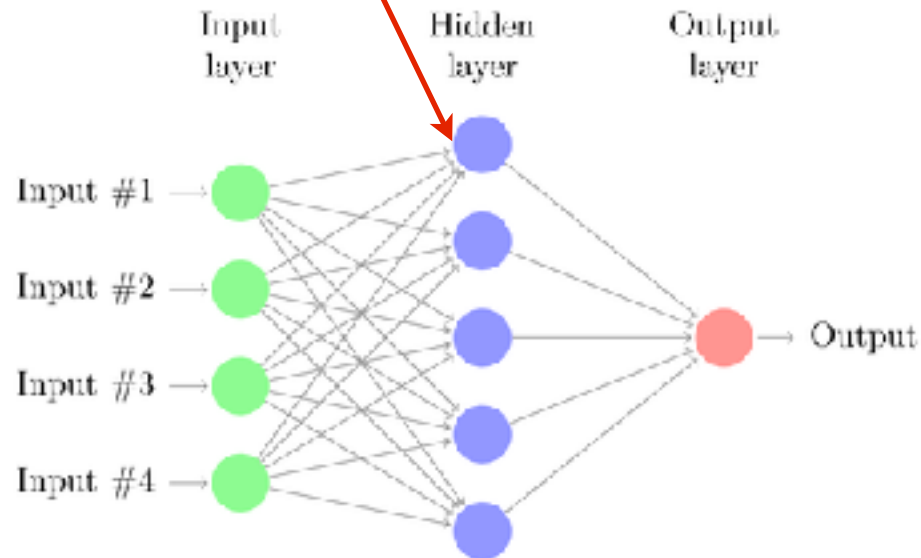
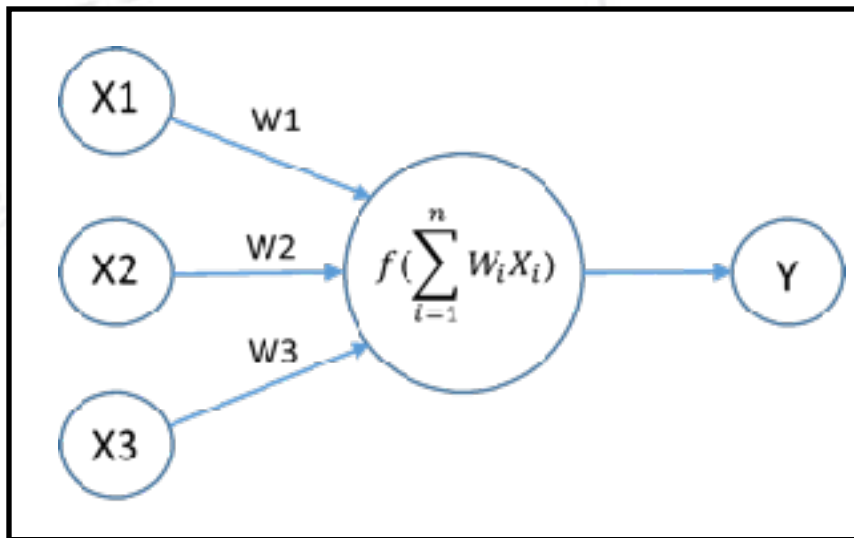
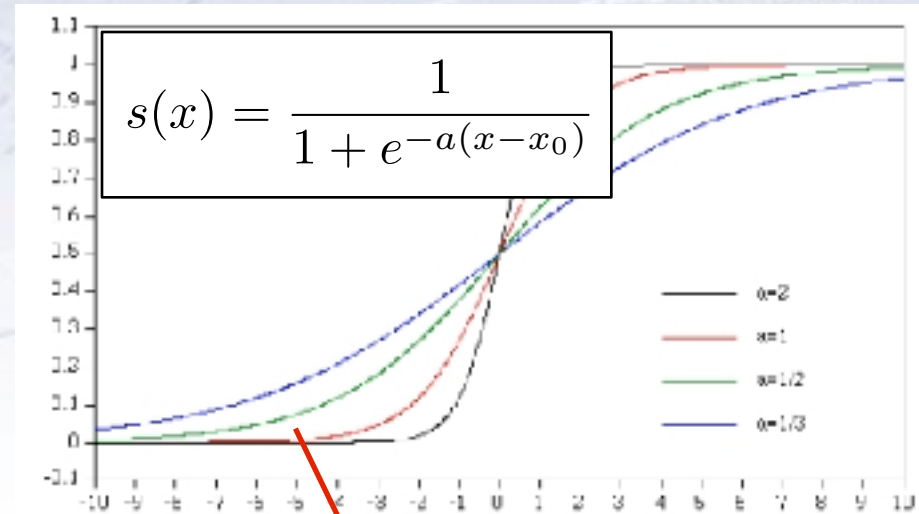


Neural Networks

Neural Networks combine the input variables using a “activation” function $s(x)$ to assign, if the variable indicates signal or background.

The simplest is a single layer perceptron:

$$t(x) = s\left(a_0 + \sum a_i x_i\right)$$



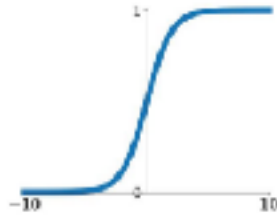
Activation Functions

There are many different activation functions, some of which are shown below. They have different properties, and can be considered a HyperParameter.

Activation Functions

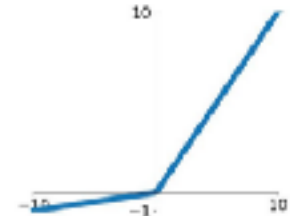
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



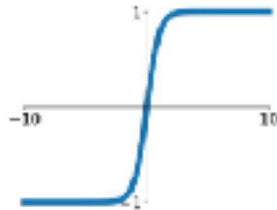
Leaky ReLU

$$\max(0.1x, x)$$



tanh

$$\tanh(x)$$

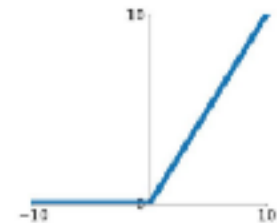


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

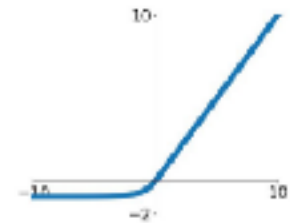
ReLU

$$\max(0, x)$$



ELU

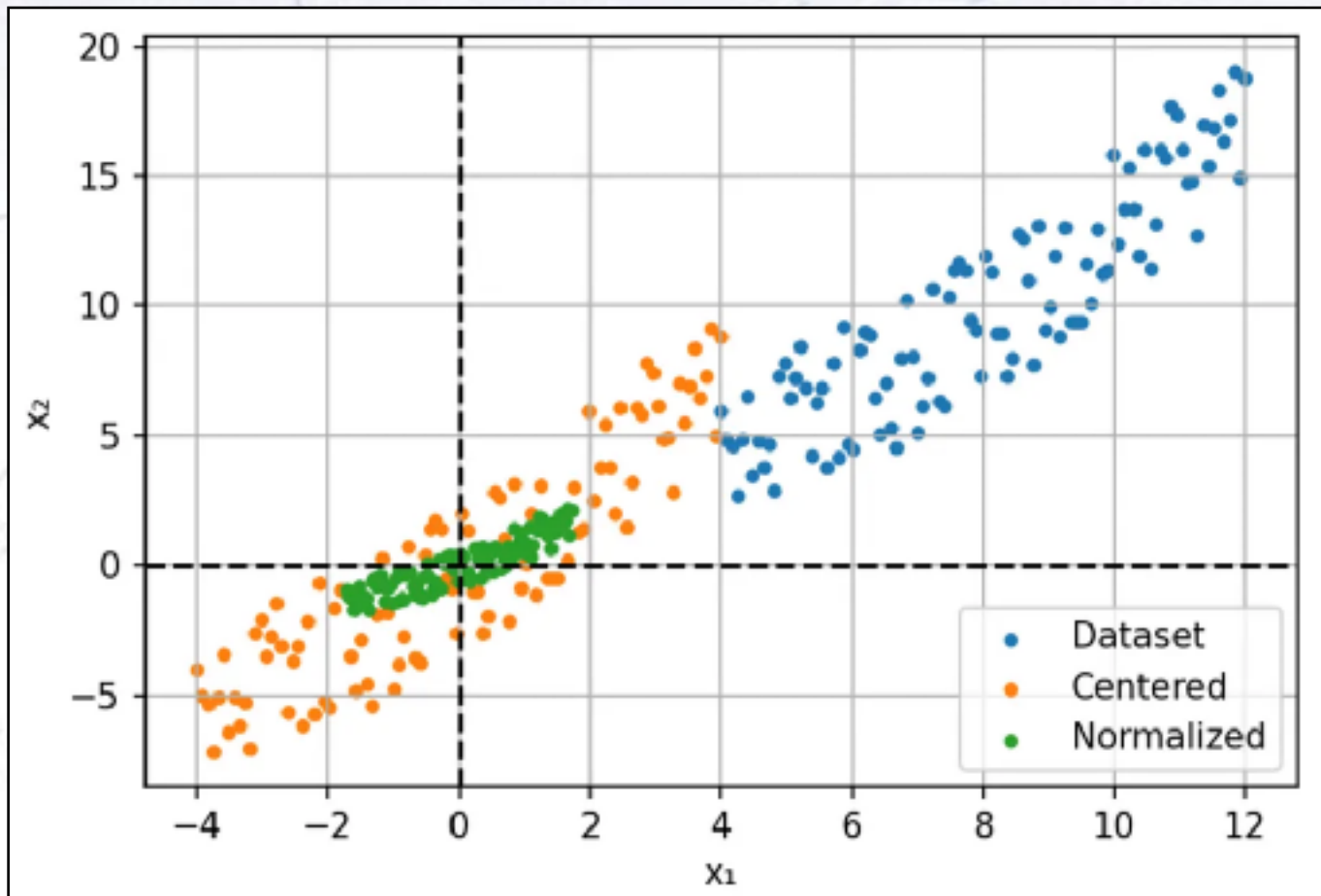
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



For a more complete list, check: https://en.wikipedia.org/wiki/Activation_function

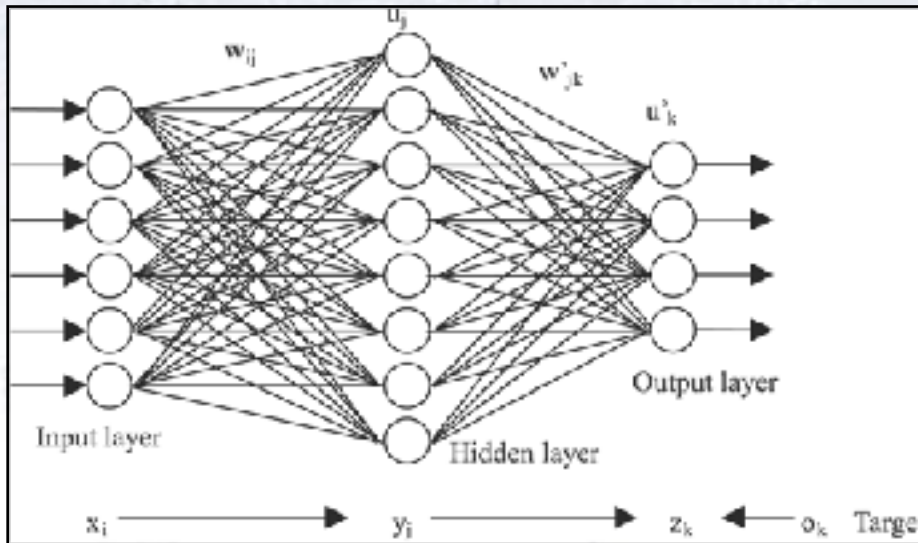
Normalising Inputs

While tree based learning is invariant to (transformations of) distributions, Neural Networks are not. To avoid hard optimisation, vanishing/exploding gradients, and differential learning rates, one should normalise the input:



Deep Neural Networks

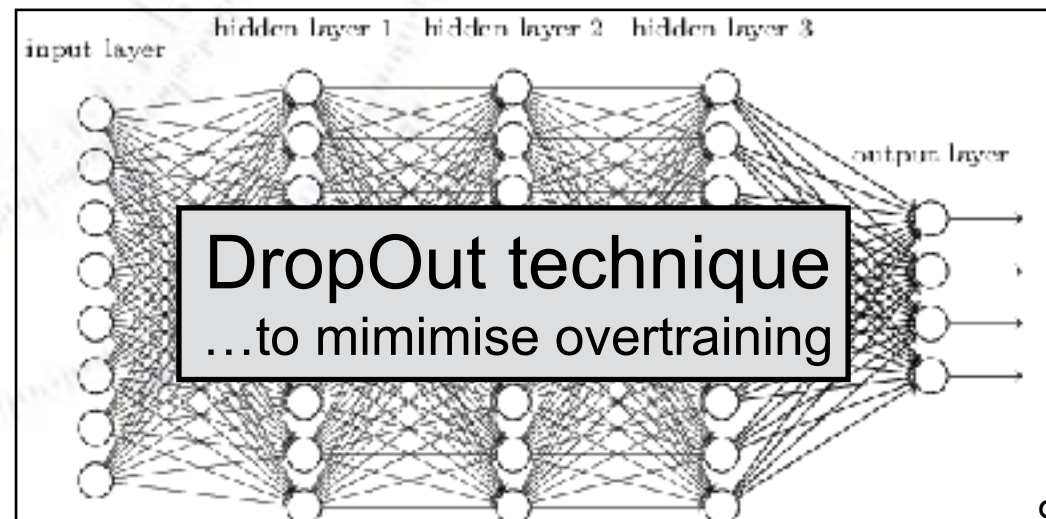
Deep Neural Networks (DNN) are simply (much) extended NNs in terms of layers!



Instead of having just one (or few) hidden layers, many such layers are introduced.

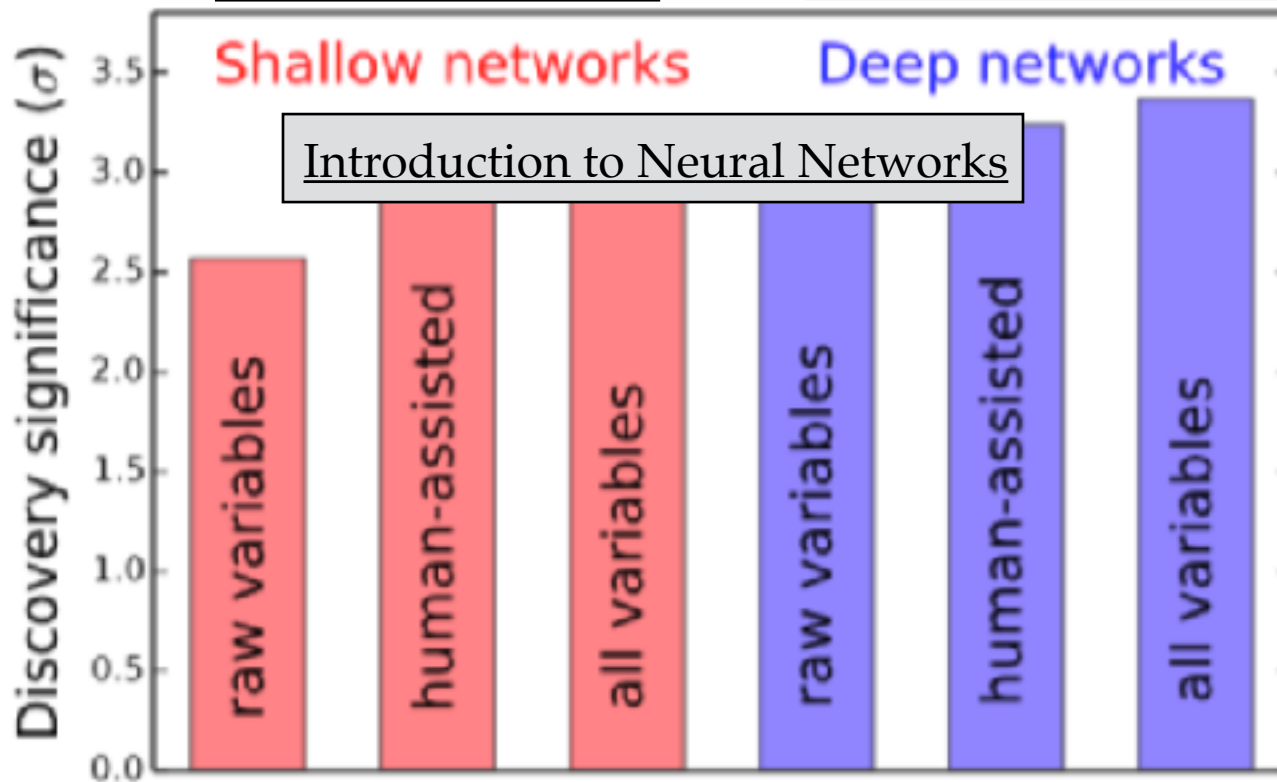
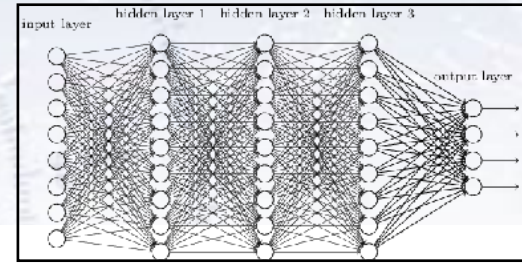
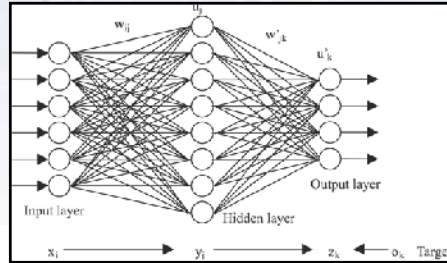
This gives the network a chance to produce key features and use them for many different specialised tasks.

Currently, DNNs can have up to millions of neurons and connections, which compares to about the **brain of a worm**.



Deep Neural Networks

Deep Neural Networks likes to get both raw and “assisted” variables:



The role of NNs

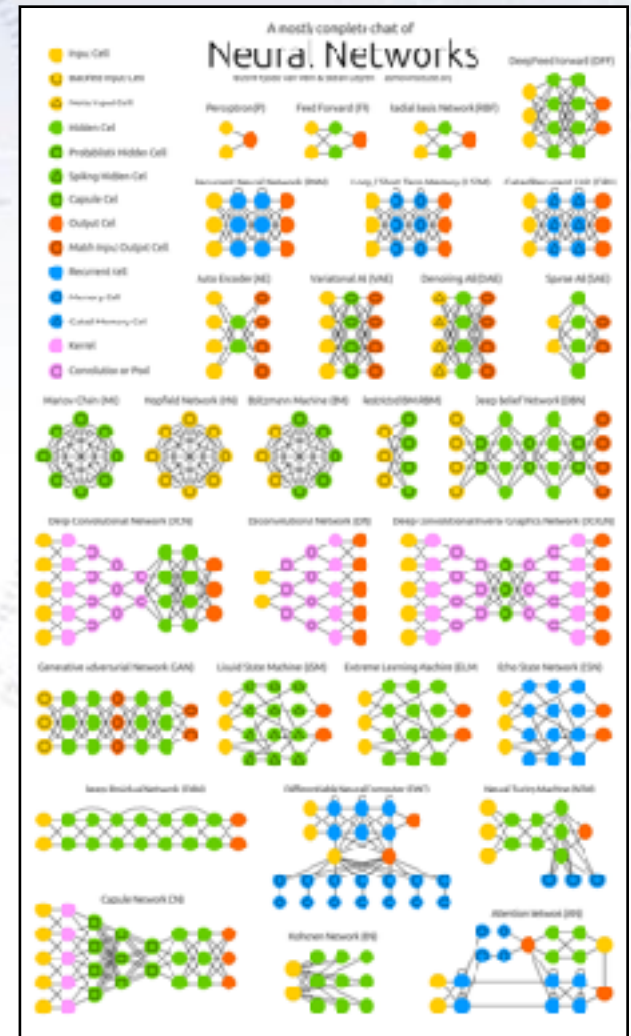
The reason why NNs play such a central role is that they are versatile:

- Recurrent NNs (for time series)
- Convolutional NNs (for images)
- Adversarial NNs (for simulation)
- Graph NNs (for geometric data)
- etc.

Unlike trees, NNs typically make the “foundation” of all the more advanced ML paradigms. However, they are harder to optimise!

This is why trees are great for simpler tasks (i.e. data that typically fits into an excel sheet [2110.01889]), while NNs are typically used for the more advanced.

Have this in mind, when you attack problems with ML - and like any other project or analysis, it is typically good to get a “rough result” fast, and then to refine it from there.



Method's (dis-)advantages

Another comparison is done in Elements of Statistical Learning II (ESL II), where linear methods are not included.

As can be seen, Neural Networks are “difficult” in almost all respects, but performant.

For trees, the case is almost the opposite.

However, I don't agree with the evaluation of the predictive power of trees.

At least not for normal structured data.

For tabular data, I disagree!

Characteristic	Neural Nets	SVM	Trees	MARS	k-NN, Kernels
Natural handling of data of “mixed” type	▼	▼	▲	▲	▼
Handling of missing values	▼	▼	▲	▲	▲
Robustness to outliers in input space	▼	▼	▲	▼	▲
Insensitive to monotone transformations of inputs	▼	▼	▲	▼	▼
Computational scalability (large N)	▼	▼	▲	▲	▼
Ability to deal with irrelevant inputs	▼	▼	▲	▲	▼
Ability to extract linear combinations of features	▲	▲	▼	▼	◆
Interpretability	▼	▼	◆	▲	▼
Predictive power	▲	▲	▼	◆	▲

...and others do too [<https://arxiv.org/abs/2110.01889>]

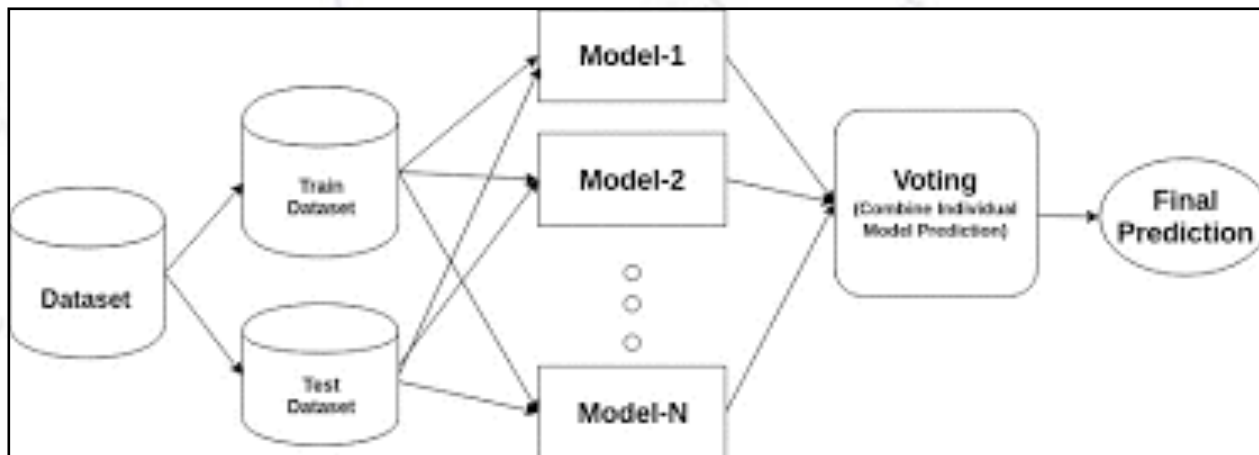
From ESL II, Chapter 10.7

Ensemble method

Different methods have different advantages, and for that reason the very best performance is often obtained by “ensemble methods”.

Here, several different ML methods are used on data, and subsequently their results are combined in a new “ensemble” ML algorithm (or by voting!), which benefits from all the advantages.

These have lately been the most performant methods (i.e. winning competitions). However, they are cumbersome (you have to optimise many methods), and typically a single method reaches close to the information limit.



A faded nautical chart is visible in the background. It features depth contours labeled with values such as 100, 150, 200, 250, 300, and 350. A coordinate marker is present, labeled 'VAR 10°15' W'. The chart also includes some text, such as '187 BITTEN ERW TAUCHT 1875'.

Loss functions

What loss function to use?

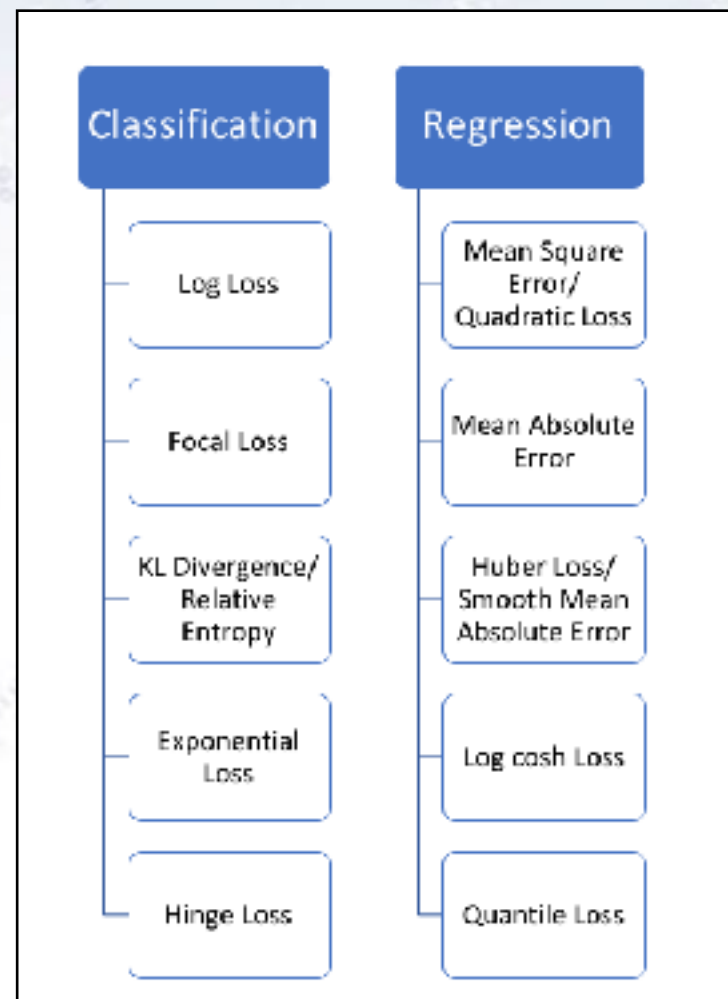
The choice of loss function depends on the problem at hand, and in particular what you find important!

In classification:

- Do you care how wrong the wrong are?
- Do you want pure signal or high efficiency?
- Does it matter what type of errors you make?

In regression:

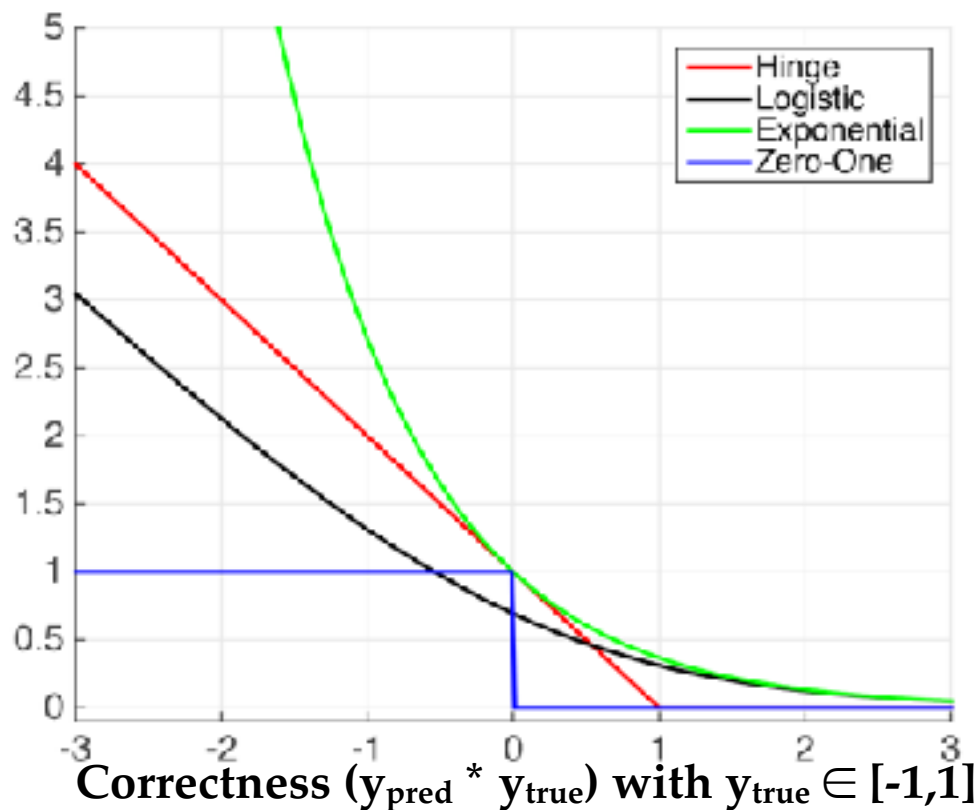
- Do you care about outliers?
- Do you care about size of outliers?
- Is core resolution vital?



What loss function to use?

The choice of loss function depends on the problem at hand, and in particular what you find important!

Loss functions for classification



Classification

Log Loss

Focal Loss

KL Divergence/
Relative
Entropy

Exponential
Loss

Hinge Loss

Regression

Mean Square
Error/
Quadratic Loss

Mean Absolute
Error

Huber Loss/
Smooth Mean
Absolute Error

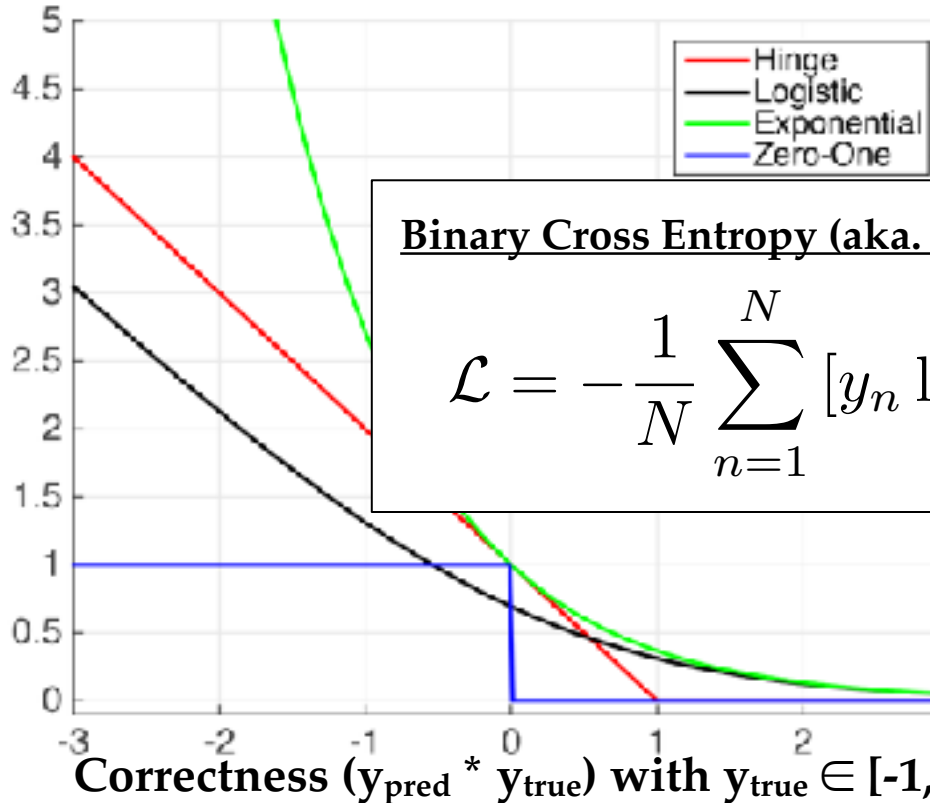
Log cash Loss

Quantile Loss

What loss function to use?

The choice of loss function depends on the problem at hand, and in particular what you find important!

Loss functions for classification



Binary Cross Entropy (aka. LogLoss or Logistic Loss):

$$\mathcal{L} = -\frac{1}{N} \sum_{n=1}^N [y_n \log \hat{y}_n + (1 - y_n) \log(1 - \hat{y}_n)]$$

Classification

Log Loss

Regression

Mean Square Error/
Quadratic Loss

Exponential Loss

Log cash Loss

Hinge Loss

Quantile Loss

Unbalanced data

If the data is unbalanced, that is if one outcome/target is much more abundant than the alternative, case has to be taken.

Example: You consider data with 19600 (98%) healthy and 400 (2%) ill patients. An algorithm always predicting “healthy” would get an accuracy score of 98%!

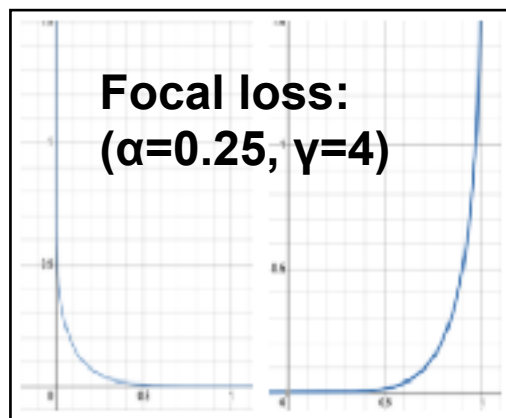
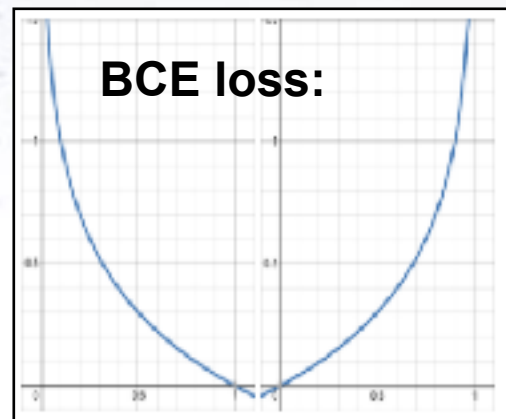
In this case, using Area Under Curve (AUC) or F1 for loss is better. An alternative is “focal loss”, which focuses on the lesser represented cases:

Binary Cross Entropy loss:

$$\mathcal{L} = -\frac{1}{N} \sum_{n=1}^N [y_n \log \hat{y}_n + (1 - y_n) \log(1 - \hat{y}_n)]$$

Focal loss:

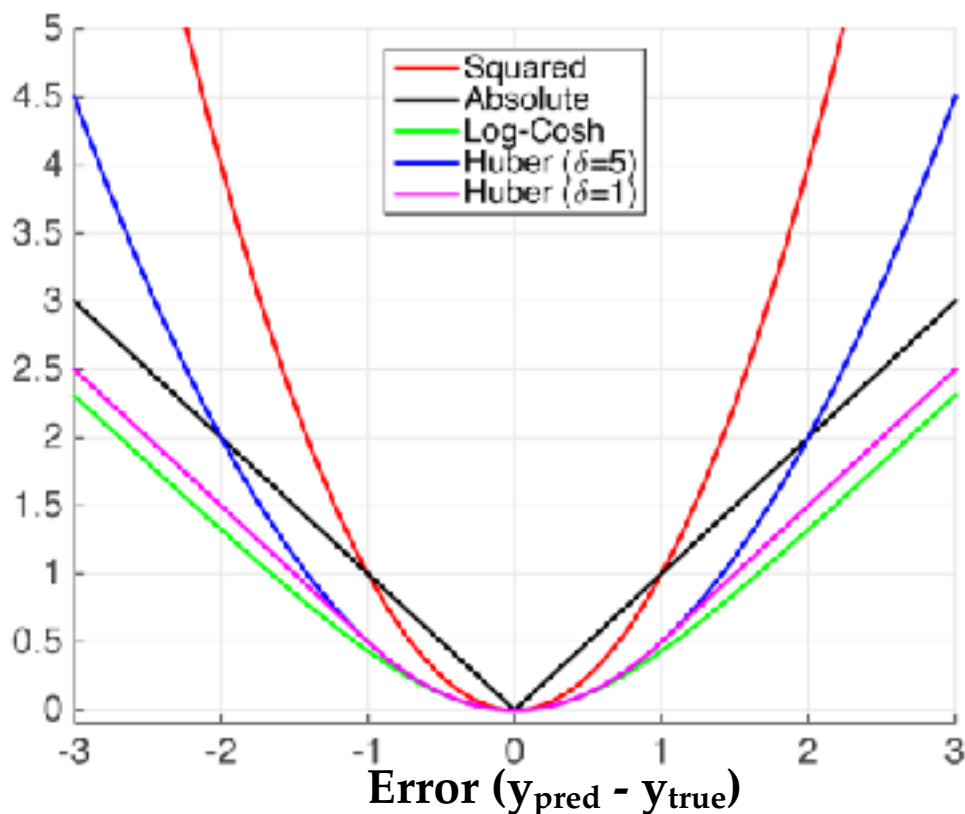
$$\mathcal{L} = -\frac{1}{N} \sum_{n=1}^N [(1 - \alpha)y_n^\gamma \log \hat{y}_n + (1 - y_n)^\gamma \log \alpha(1 - \hat{y}_n)]$$



What loss function to use?

The choice of loss function depends on the problem at hand, and in particular what you find important!

Loss functions for regression



Classification

Log Loss

Focal Loss

KL Divergence/
Relative
Entropy

Exponential
Loss

Hinge Loss

Regression

Mean Square
Error/
Quadratic Loss

Mean Absolute
Error

Huber Loss/
Smooth Mean
Absolute Error

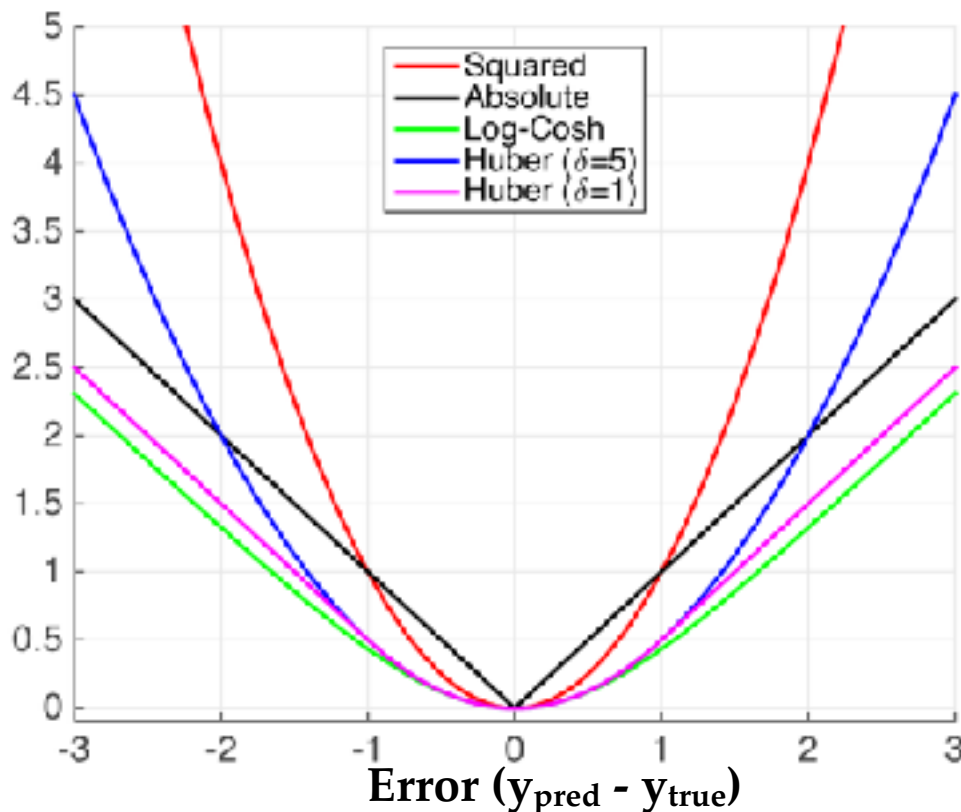
Log cosh Loss

Quantile Loss

What loss function to use?

The choice of loss function depends on the problem at hand, and in particular what you find important!

Loss functions for regression



Squared Loss:

- Most popular regression loss function
- Estimates Mean Label
- ADVANTAGE: Differentiable everywhere
- DISADVANTAGE: Sensitive to outliers

Absolute Loss:

- Also a very popular loss function
- Estimates Median Label
- ADVANTAGE: Less sensitive to noise
- DISADVANTAGE: Not differentiable at 0

Huber Loss:

- ADVANTAGE: "Best of Both Worlds" of Squared and Absolute Loss.
- DISADVANTAGE: Only once-differentiable

LogCosh Loss:

- ADVANTAGE: "Best of Both Worlds" of Squared and Absolute Loss.
- ADVANTAGE: Similar to Huber Loss, but twice differentiable everywhere.

What loss function to use?

The choice of loss function depends on the problem at hand, and in particular what you find important!

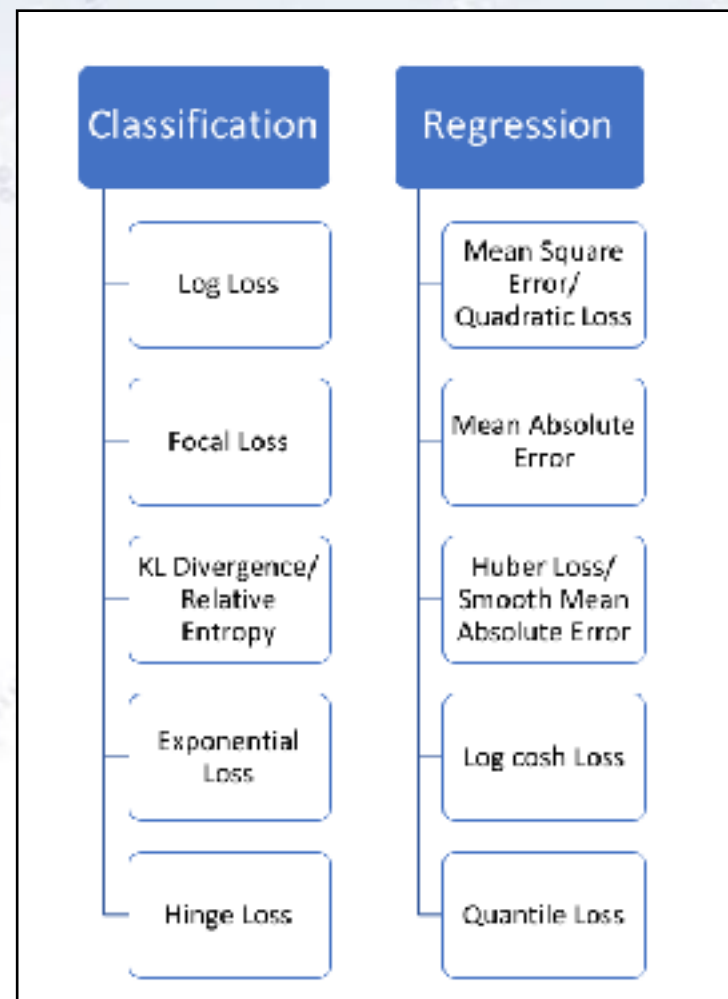
In classification:

- Do you care how wrong the wrong are?
- Do you want pure signal or high efficiency?
- Does it matter what type of errors you make?

In regression:

- Do you care about outliers?
- Do you care about size of outliers?
- Is core resolution vital?

Ultimately, the loss function should be tailored to match the wishes of the user. This is however not always that simple, as this might be hard to even know!



XGBoost algorithm

In order to “punish” complexity, the cost-function has a regularised term also:

$$\mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

$$\text{where } \Omega(f) = \gamma T + \frac{1}{2} \lambda \|w\|^2$$

Table 1: Comparison of major tree boosting systems.

System	exact greedy	approximate global	approximate local	out-of-core	sparsity aware	parallel
XGBoost	yes	yes	yes	yes	yes	yes
pGBRT	no	no	yes	no	no	yes
Spark MLlib	no	yes	no	no	partially	yes
H2O	no	yes	no	no	partially	yes
scikit-learn	yes	no	no	no	no	no
R GBM	yes	no	no	no	partially	no

XGBoost algorithm

In order to “punish” complexity, the cost-function has a regularised term also:

$$\mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

$$\text{where } \Omega(f) = \gamma T + \frac{1}{2} \lambda \|w\|^2$$

Table 1: Comparison of major tree boosting systems.

System	exact greedy	approximate global	approximate local	out-of-core	sparsity aware	parallel
XGBoost						
pGBRT						
Spark MLlib						
H2O						
scikit-learn	yes	no	no	no	no	no
R GBM	yes	no	no	no	partially	no

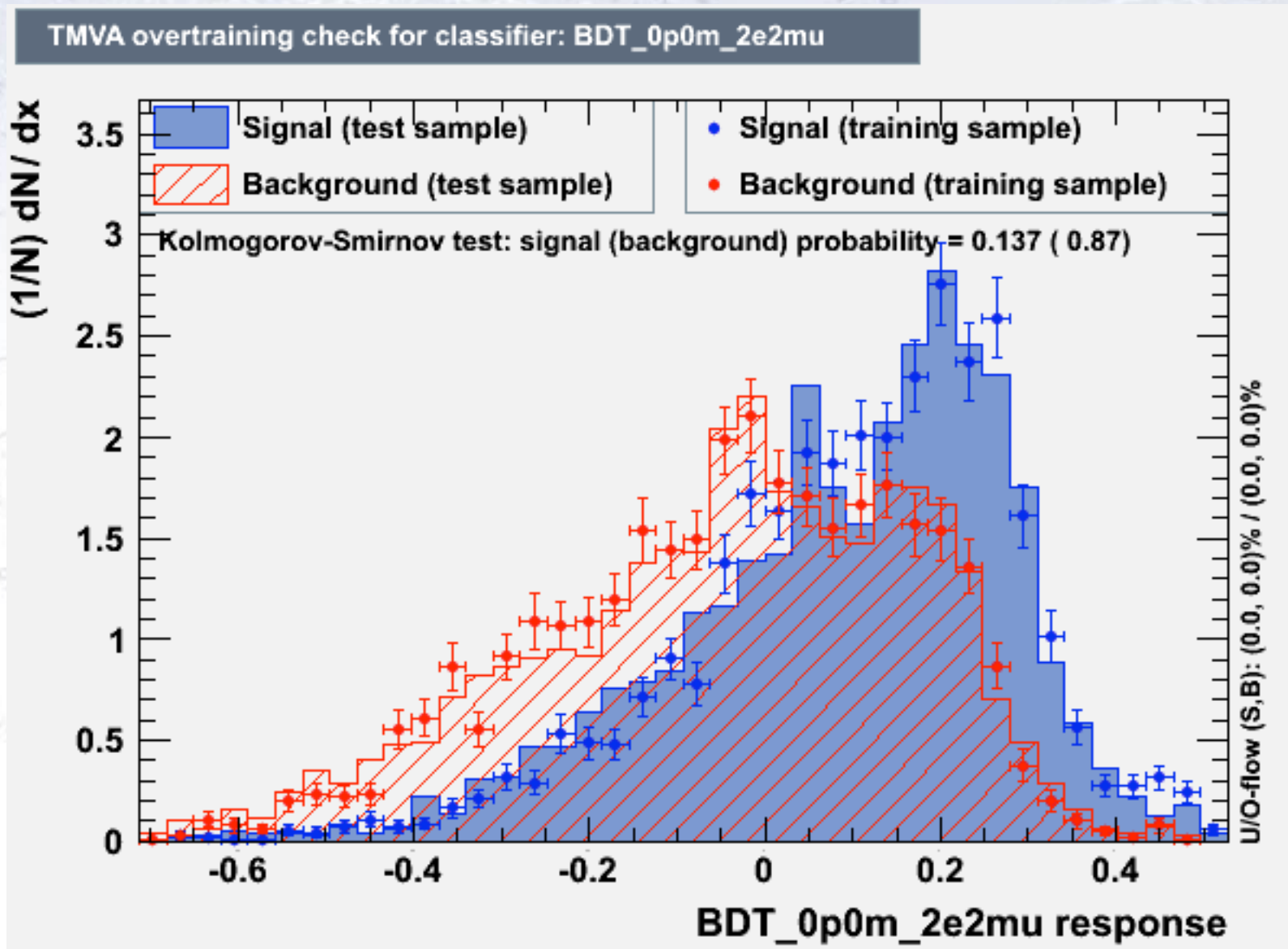
Generally, all constraints or priors should be included into the model through additions to the loss function.

A faded nautical chart serves as the background. It features depth contours in fathoms, with labels such as 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200, 210, 220, 230, 240, 250, 260, 270, 280, 290, 300, 310, 320, 330, 340, 350, 360, 370, 380, 390, 400, 410, 420, 430, 440, 450, 460, 470, 480, 490, 500, 510, 520, 530, 540, 550, 560, 570, 580, 590, 600, 610, 620, 630, 640, 650, 660, 670, 680, 690, 700, 710, 720, 730, 740, 750, 760, 770, 780, 790, 800, 810, 820, 830, 840, 850, 860, 870, 880, 890, 900, 910, 920, 930, 940, 950, 960, 970, 980, 990, 1000. A red crosshair is positioned at approximately 50° 15' N latitude and 10° 15' W longitude. The chart also includes various navigational symbols and text, such as 'WACHT' and '187 BITTEN ERWACHT 1879'.

Train, Validation & Test

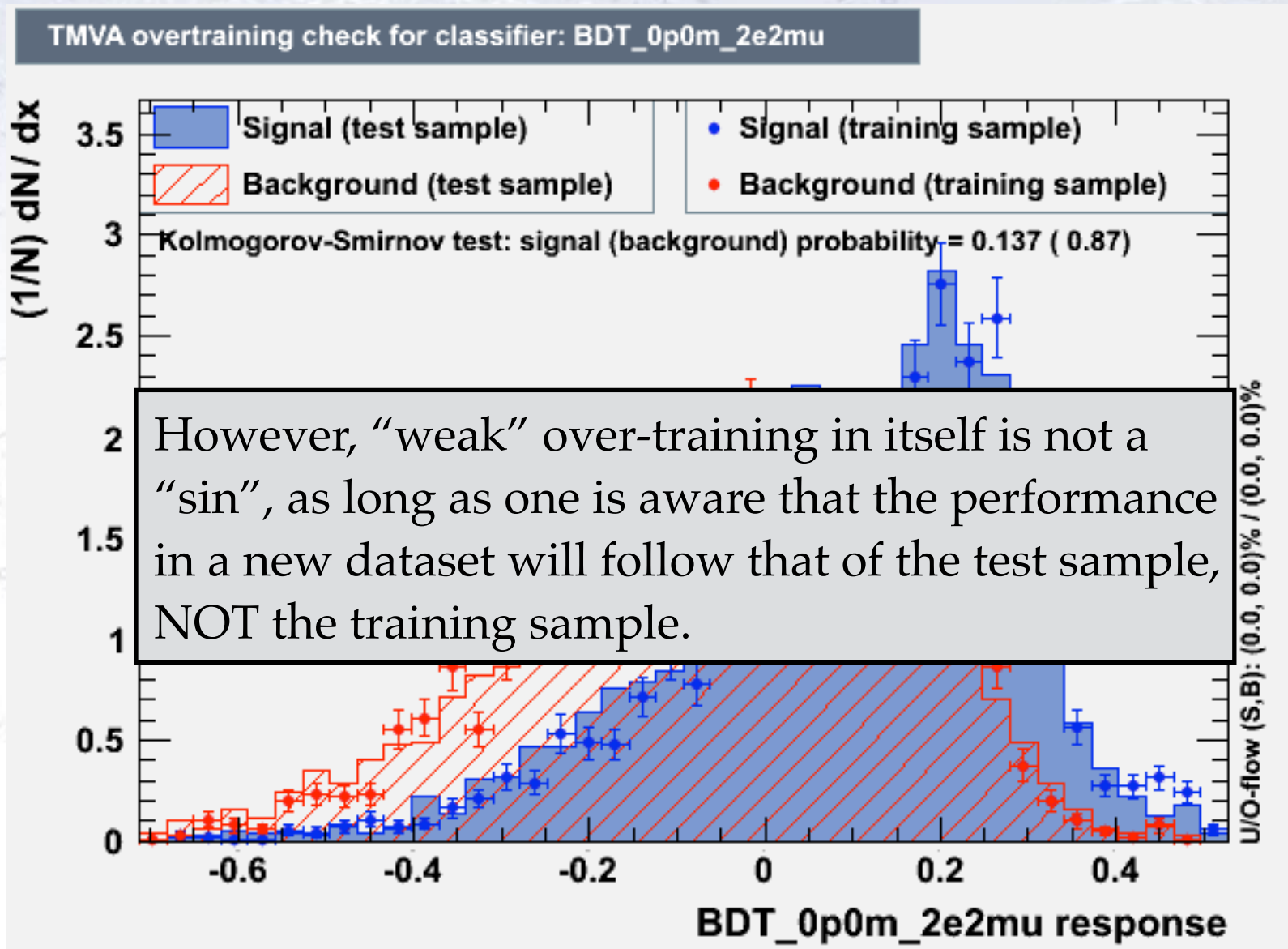
Test for simple over-training

In order to test for overtraining, half the sample is used for training, the other for testing:



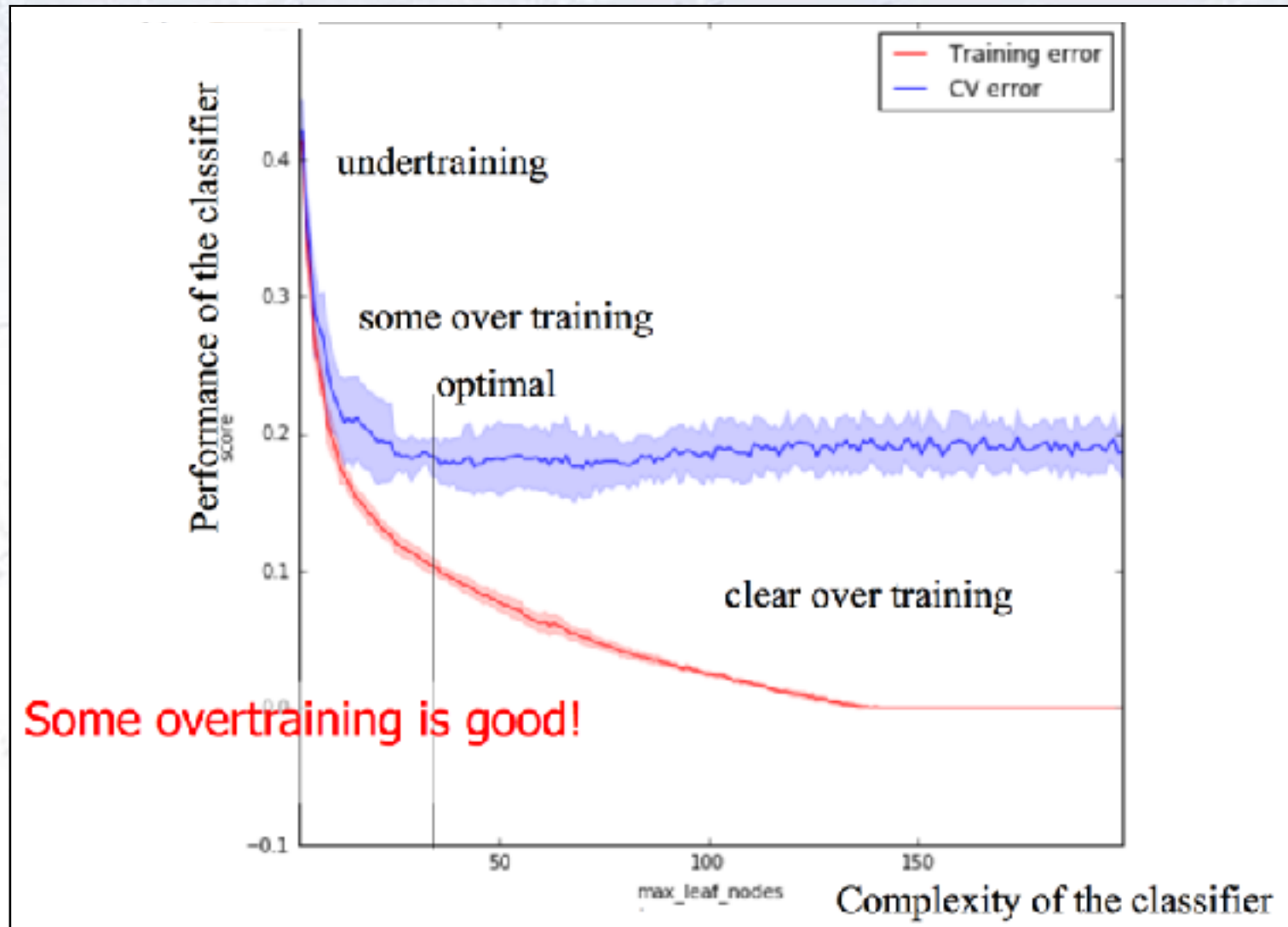
Test for simple over-training

In order to test for overtraining, half the sample is used for training, the other for testing:



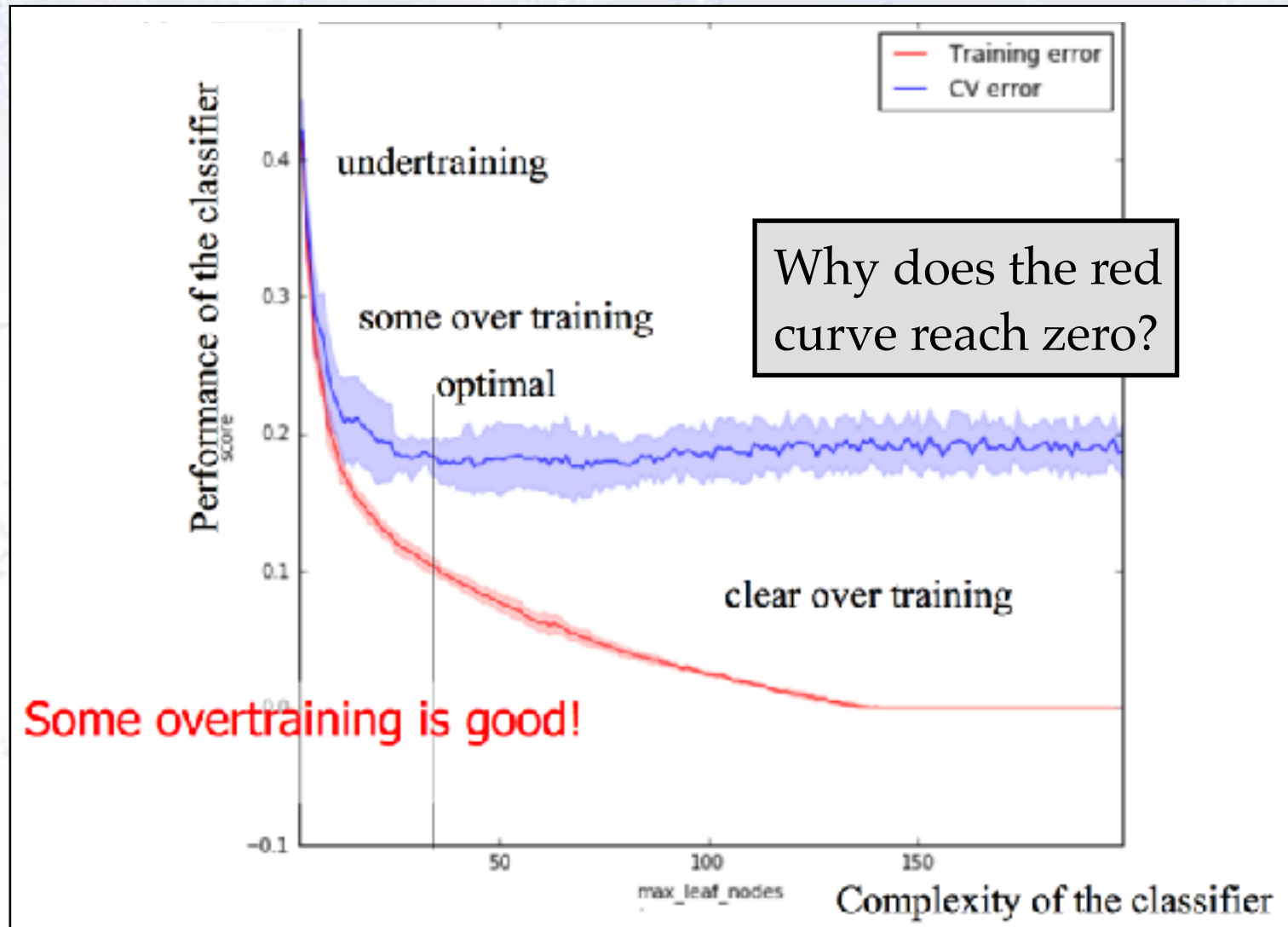
Real overtraining

The “real” limit of overtraining, is when the (Cross) Validation (CV) error starts to grow!



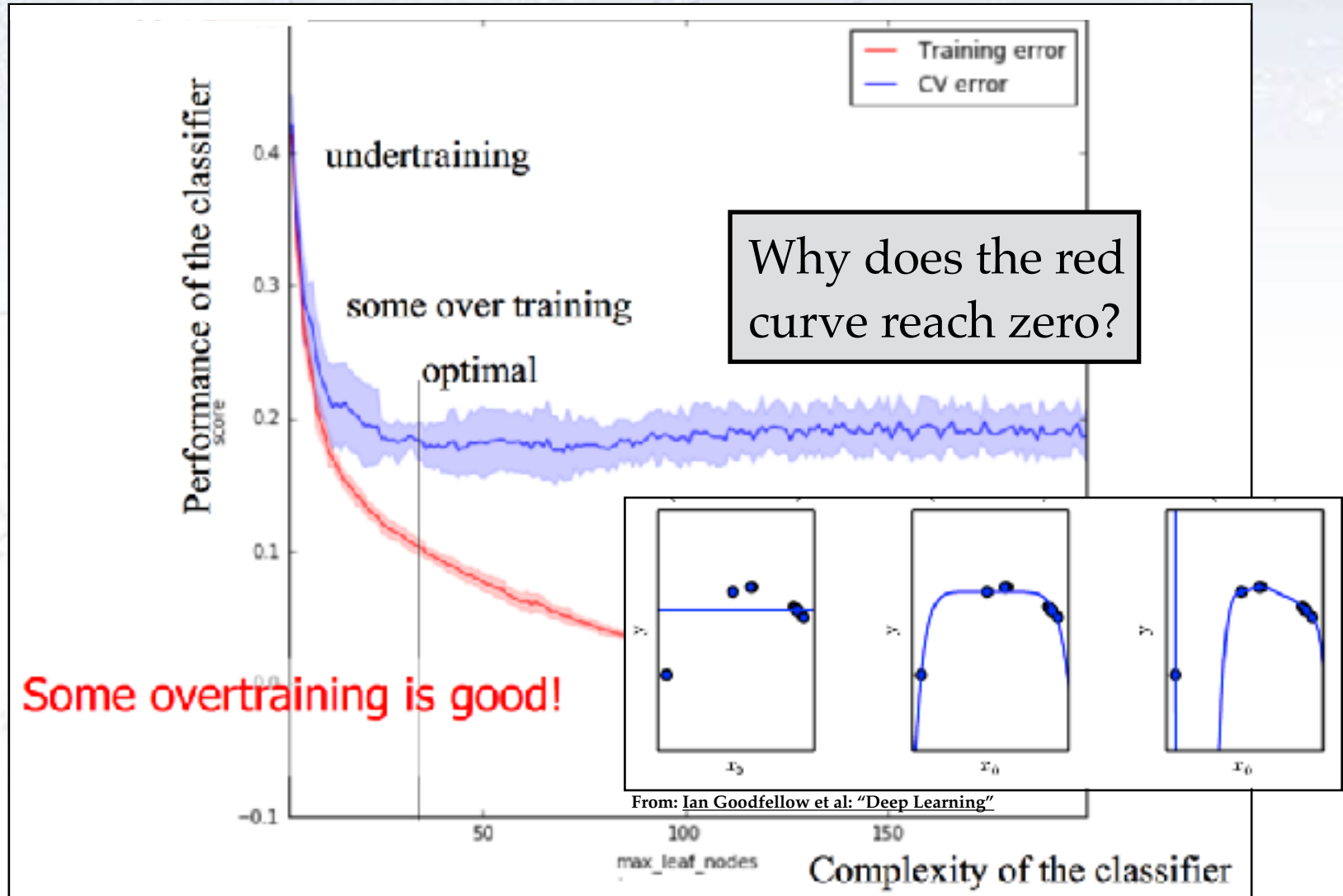
Real overtraining

The “real” limit of overtraining, is when the (Cross) Validation (CV) error starts to grow!



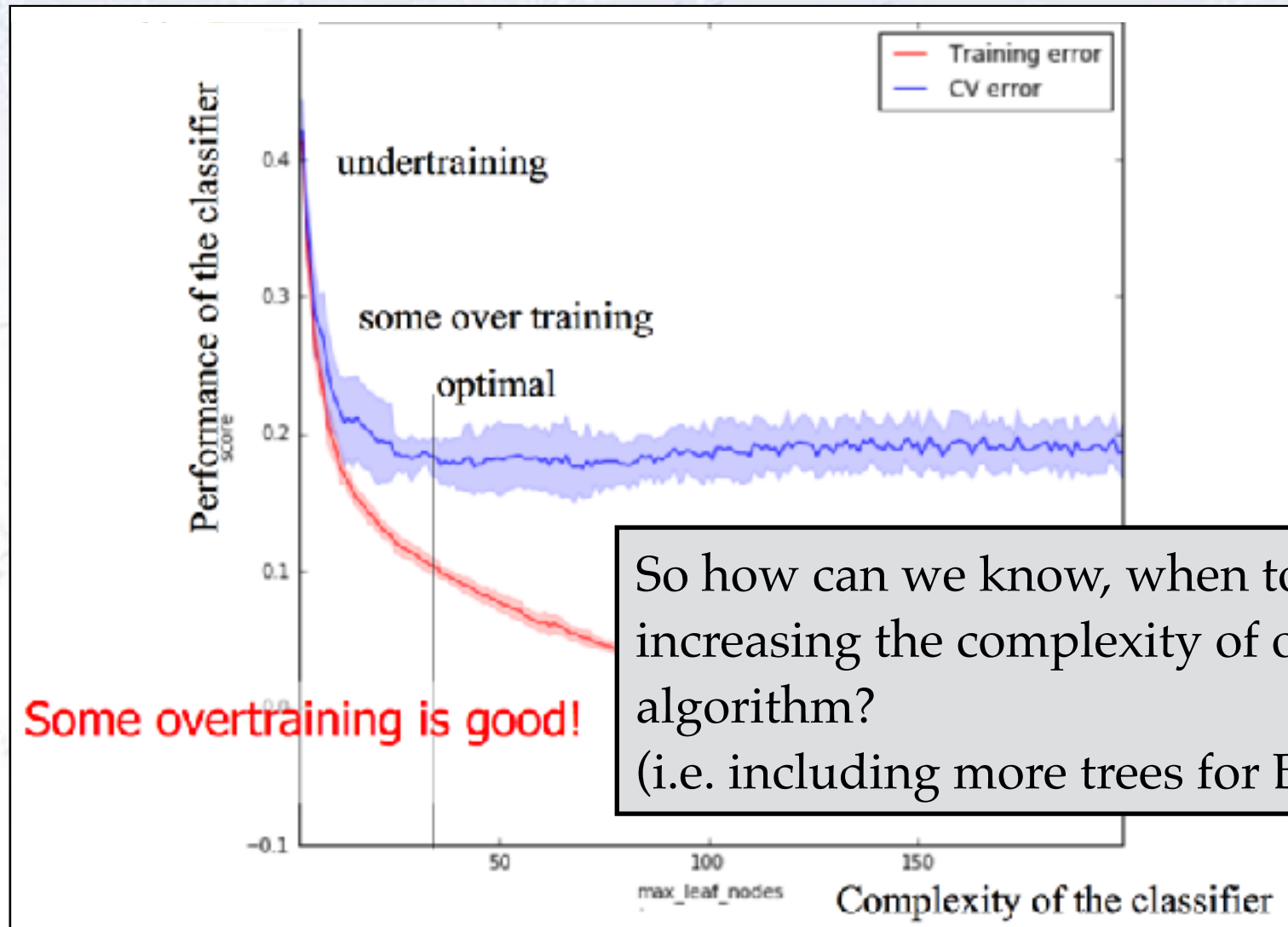
Real overtraining

The “real” limit of overtraining, is when the (Cross) Validation (CV) error starts to grow!



Real overtraining

The “real” limit of overtraining, is when the (Cross) Validation (CV) error starts to grow!





Dividing Data

How to “use” your data?

If you train you algorithm on all data, you will not recognise overtrain, nor what the expected performance on new data will be. Thus we divide the data into:

Train Dataset

- Set of data used for **learning** (by the model), that is, to fit the parameters to the machine learning model using **stochastic gradient descent**.

Valid Dataset

- Set of data used to provide an **unbiased evaluation of intermediate models** fitted on the training dataset while tuning model parameters and hyperparameters, and also for selecting input features.

Test Dataset

- Set of data used to provide an **unbiased evaluation of a final model** fitted on the training dataset.



How to do the division?

You can of course do this yourself with your own code, but there are specially prepared functions for the task:

Scikit-Learn method:

```
from sklearn.model_selection import train_test_split
X_train, X_rem, y_train, y_rem = train_test_split(X, y, train_size=0.8)
X_valid, X_test, y_valid, y_test = train_test_split(X_rem, y_rem, test_size=0.5)
```

Fast ML method:

```
from fast_ml.model_development import train_valid_test_split
X_train, y_train, X_valid, y_valid, X_test, y_test =
train_valid_test_split(df, target = '?', train_size=0.8, valid_size=0.1, test_size=0.1)
```

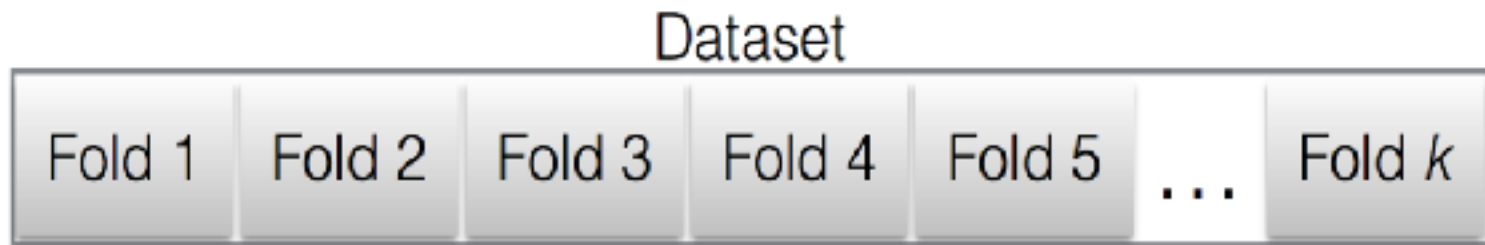
There are a few important things to remember:

- Always do the data cleaning, selecting, weighting, etc. **before** splitting!
- If there is “more than enough” data, then use **less than the total**.
- If there is “a little too little” data, then use **cross validation (next)**.

k-fold Cross Validation

In case your data set is not that large (and perhaps anyhow), one can train on most of it, and then test on the remaining $1/k$ fraction.

This is then repeated for each fold... CPU-intensive, but easily parallelisable and smart especially for small data samples.



- ▶ Split the dataset into k randomly sampled independent subsets (folds).
- ▶ Train classifier with k-1 folds and test with remaining fold.
- ▶ Repeat k times.

Getting an uncertainty estimate

The k-fold cross validation (CV) does not only allow you to train on almost all $(1 - (1/k))$ and test on all the data, but also has a two additional advantages:

- If you consider the performance (“Error” below) on each fold, then you can also calculate the uncertainty on the performance.
- Since you can test on all data, the uncertainty on the loss estimate goes down.



Why use CV?

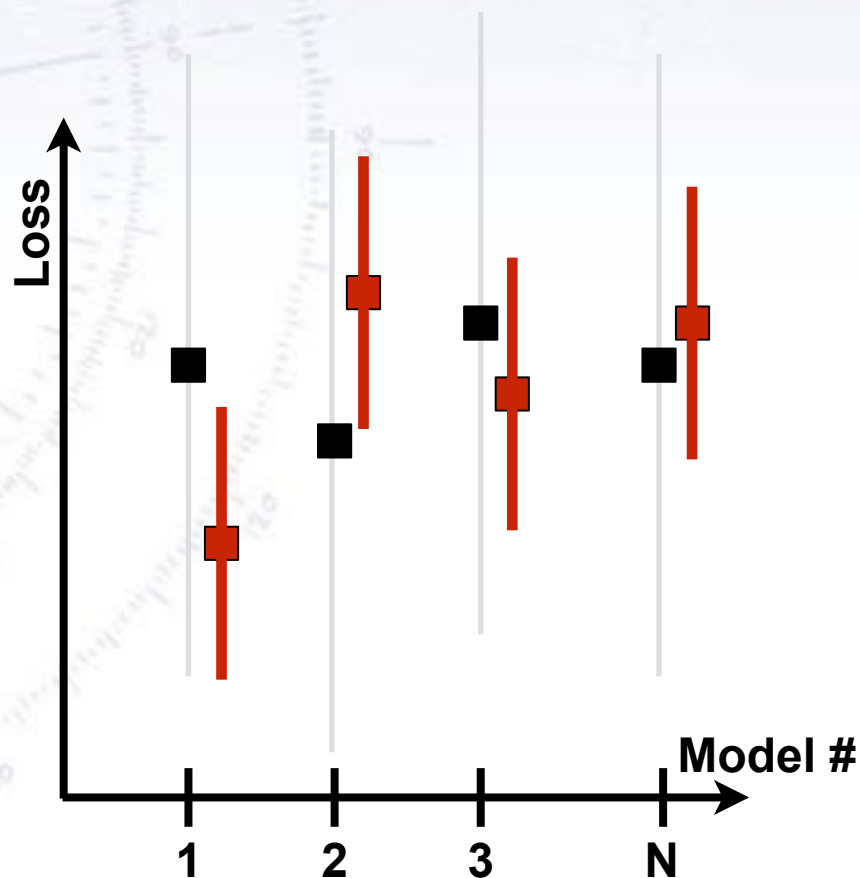
The k-fold cross validation (CV) allows you to get a better error estimate and knowledge of the uncertainty.

Imagine that you train N different models (different type, HPs, training, etc.), and that you get results as shown:

You conclude that model #2 is best. However, you don't know, that the uncertainties are rather large, because your test sample (20%) is small!

Then you do 5-fold CV... and get a more accurate evaluation with smaller uncertainties (by factor $1/\sqrt{5}$).

Now you conclude, that model #1 is the best... and that model #2 is worst!



Why use CV?

The k-fold cross validation (CV) allows you to get a better error estimate and knowledge of the uncertainty.

Imagine that you train N different models (different type, HPs, training, etc.), and that you get results as shown:

You conclude

However, you

uncertainties

your test set

Then you do

a more accurate

smaller uncertainties

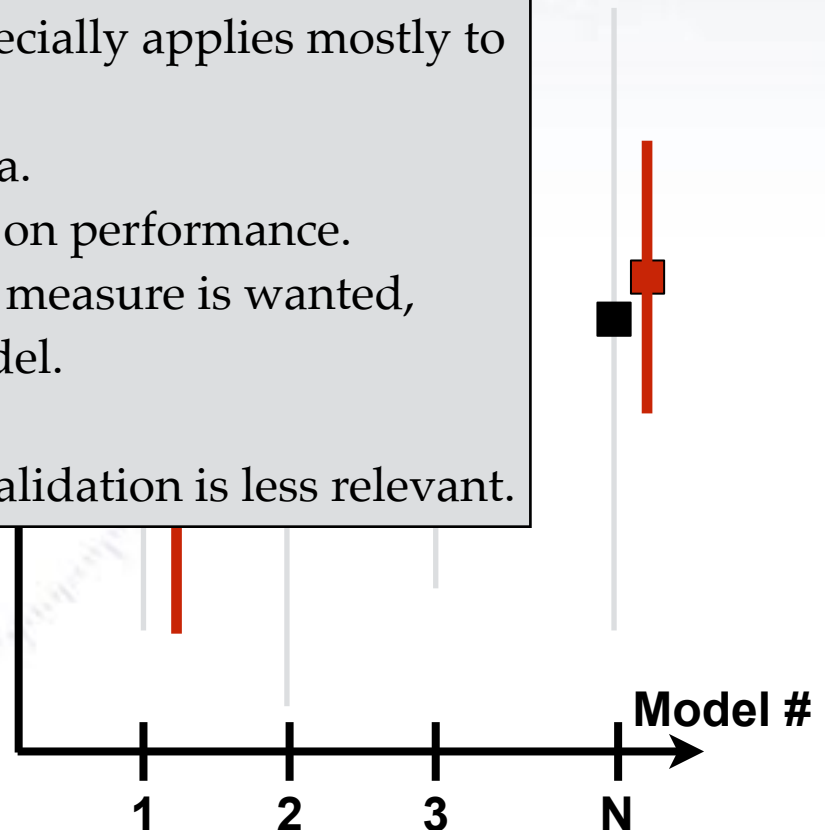
(by factor $1/\sqrt{5}$).

Now you conclude, that model #1 is the best... and that model #2 is worst!

Note that Cross Validation especially applies mostly to three cases:

- When there is little (test) data.
- When you want uncertainty on performance.
- When accurate performance measure is wanted, e.g. to find the very best model.

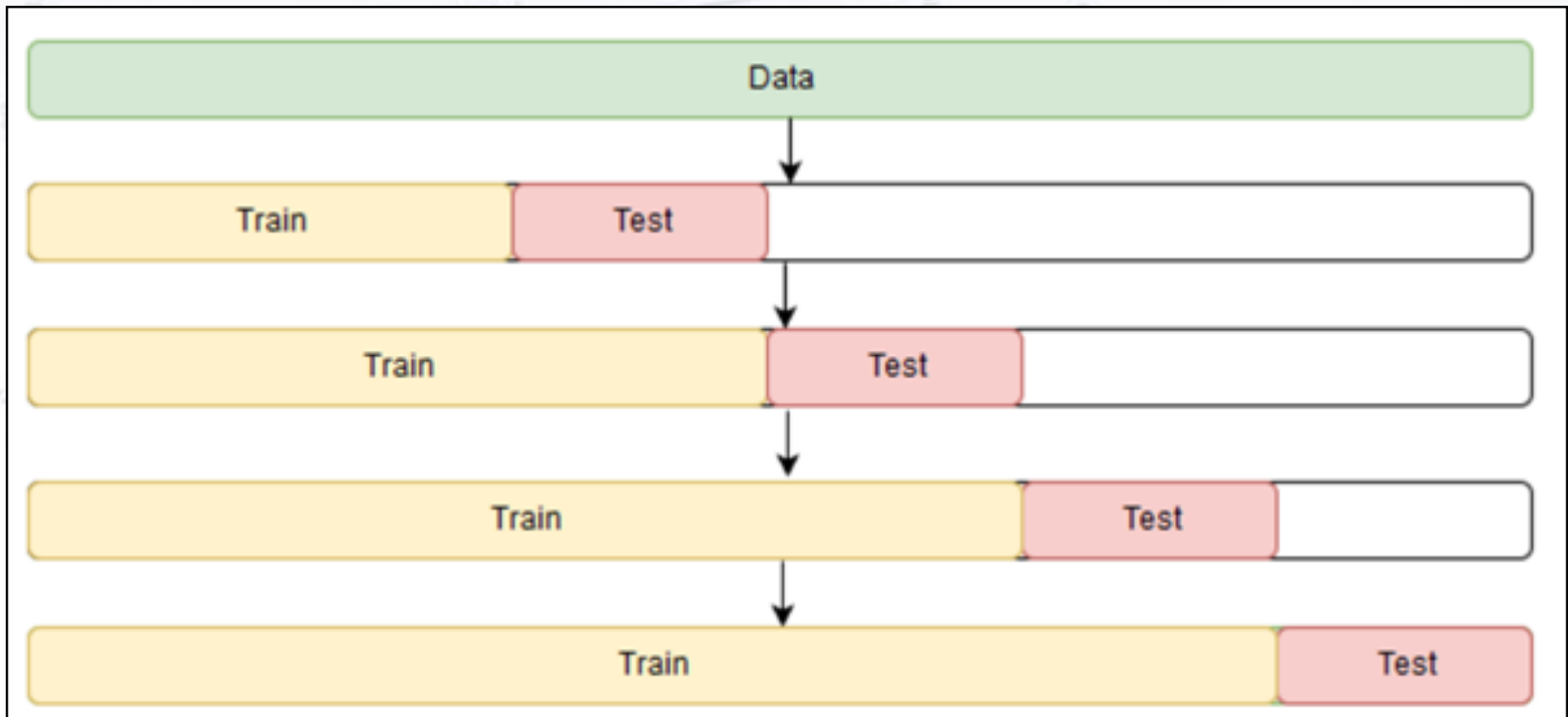
At very high statistics, Cross Validation is less relevant.



CV for time series

Special care has to be taken, when doing Cross Validation for time series, as one should ensure that **training is done only on data from the past, not the future!**

The figure below illustrates the principle. One should choose a certain data period, and put the test period immediately after, and then shift this setup.





Preprocessing Data

When data is imperfect

So far, we have looked at “perfect” data, i.e. data without any flaws in it. However, real world datasets are hardly ever “perfect”, but contains flaws that makes preprocessing imperative.

Effects may be (non-exhaustive list):

- NaN-values and "Non-values" (i.e. -9999)
- Wild outliers (i.e. values far outside the typical range)
- Shifts in distributions (i.e. part of data having a different mean/width/etc.)
- Mixture of types (i.e. numerical and text, from something humans filled out)

It is also important to consider, if entries are missing...

1. **Randomly** (in which case there should be no bias from omitting) or
2. **Following some pattern** (in which case there could be problems!).

In case of NaN values, we might simply decide to drop the variable column or entry row, requiring that all variables/entries have reasonable values.

Alternatively, we might insert “imputed” values instead, saving statistics.

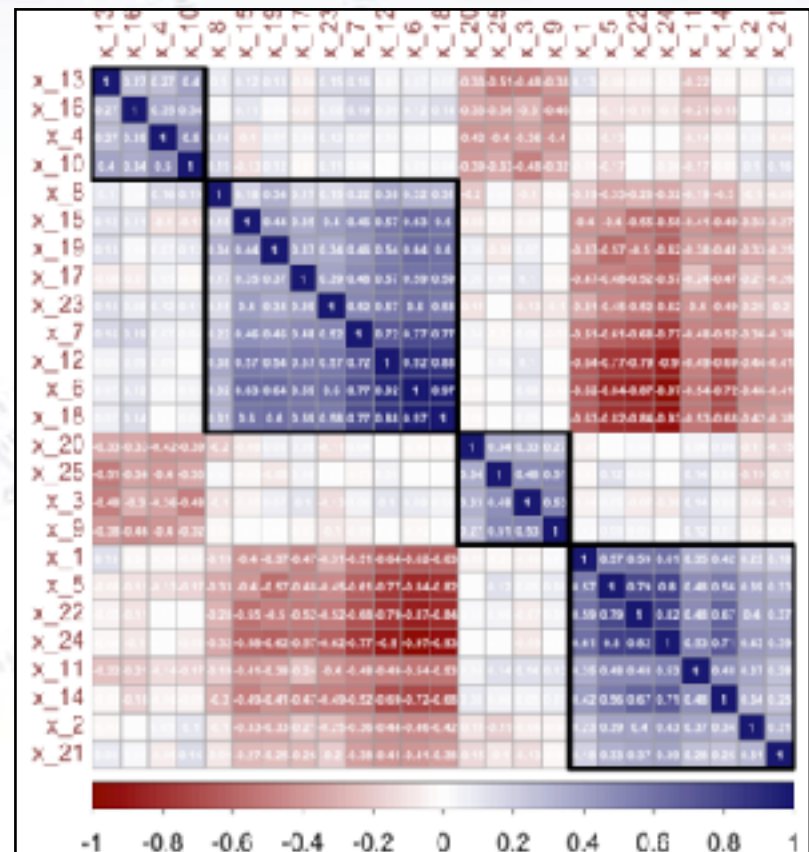
NaN-values tend to correlate

It is often seen, that several variables have the same source, and thus their NaN occurrence might be correlated with each other.

This can be tested by substituting 0's for numerical values and 1's for NaN values. By considering the correlation matrix of these substitute 0/1 values, one gets a pretty clear picture.

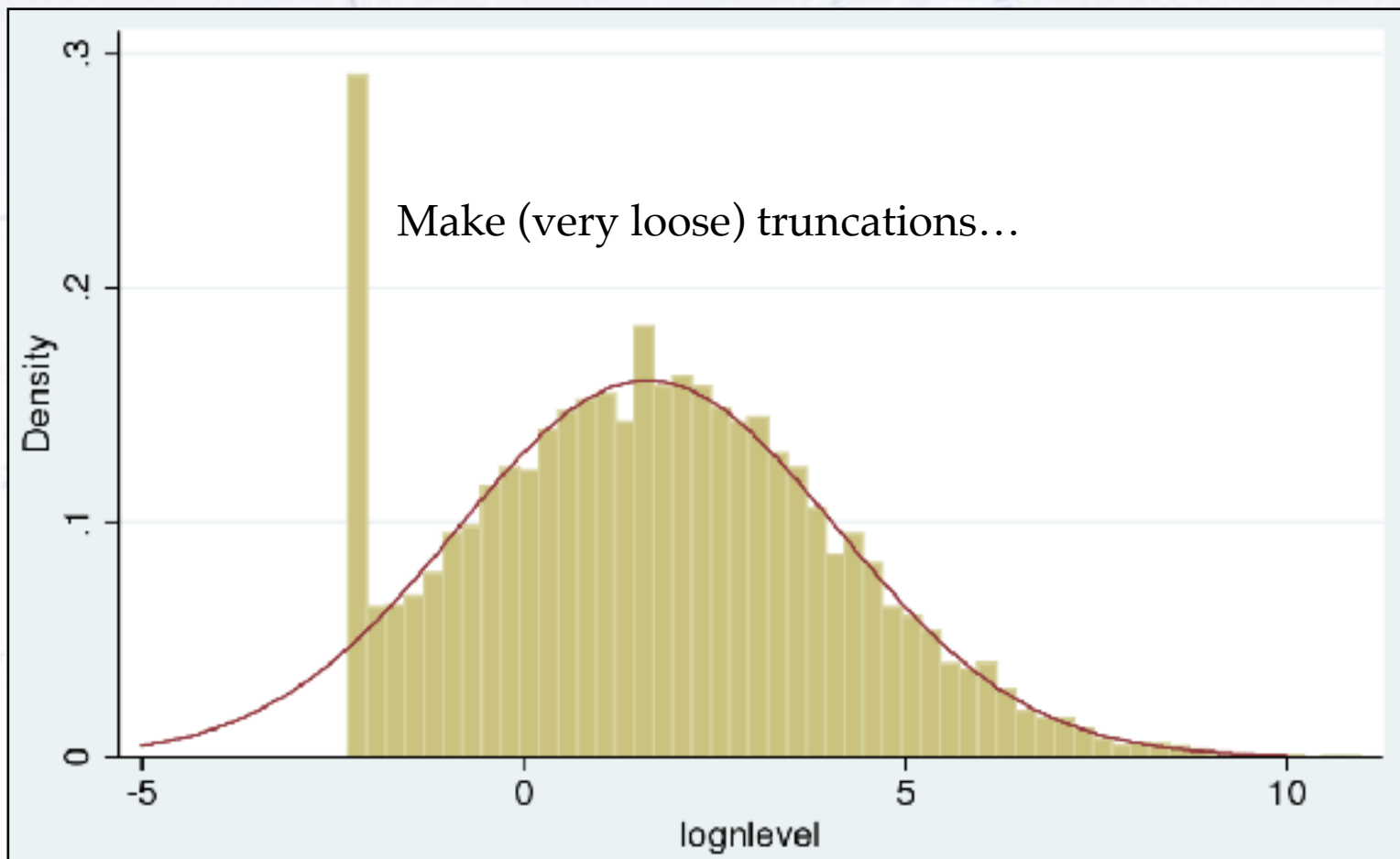
Typically, some entries are 100% correlated, as the source of these variables is shared.

Based on this information, one can better decide how to deal with these variables.



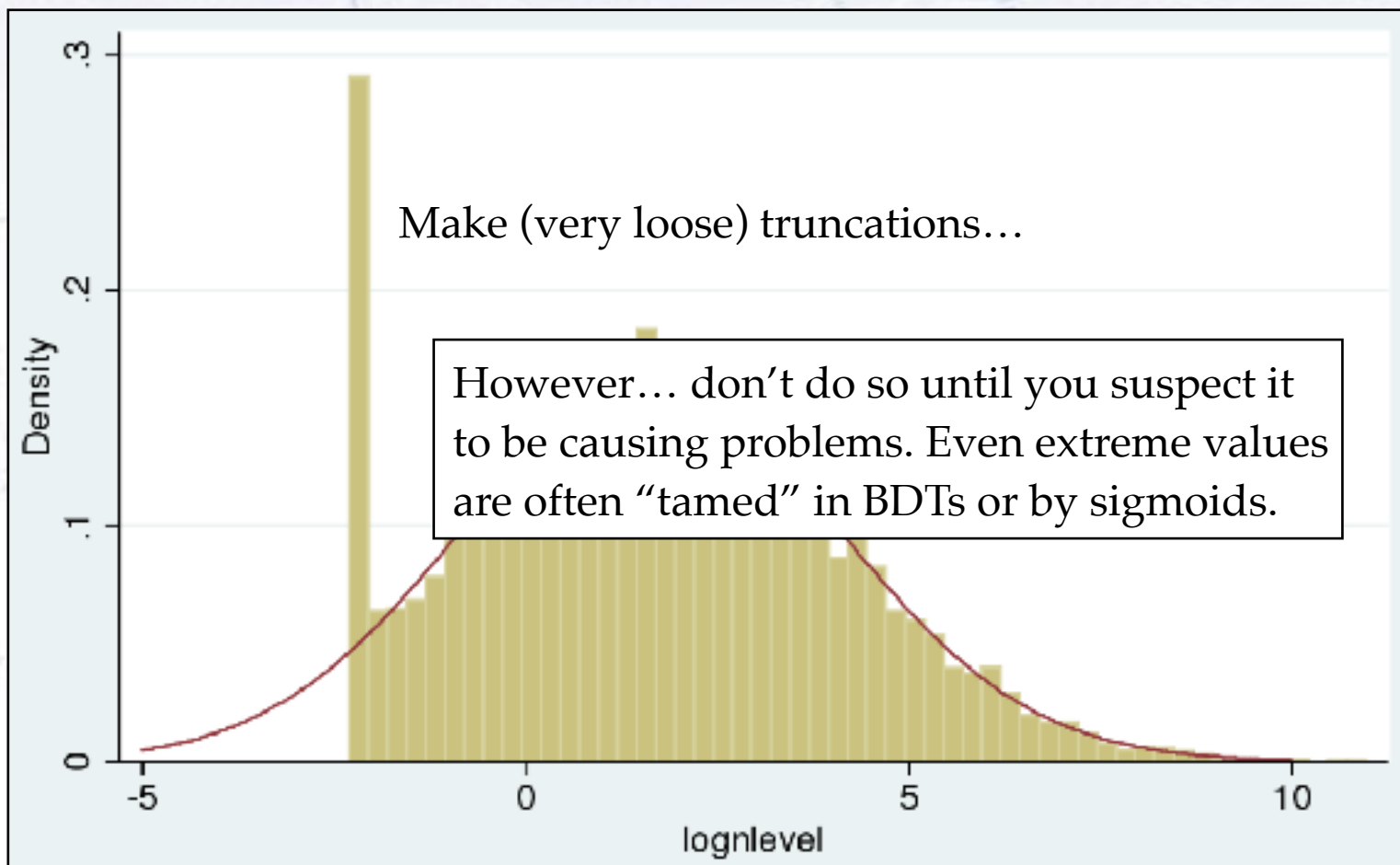
How to deal with outliers?

Sometimes, (a few?) entries take on extreme values, which ruin either the NN performance, or the transformation applied first (and then the performance). How to deal with that?



How to deal with outliers?

Sometimes, (a few?) entries take on extreme values, which ruin either the NN performance, or the transformation applied first (and then the performance). How to deal with that?



Conclusions

No matter what you plan to do with data, my first advice is always:

Print & Plot

This is your first assurance, that you even remotely know what the data contains, and your first guard against nasty surprises.

Also, working with others (from know-nothings to domain experts) you will be required to show the input, and assuring that it is valid and makes sense.

Remember to do so in all your ML work...

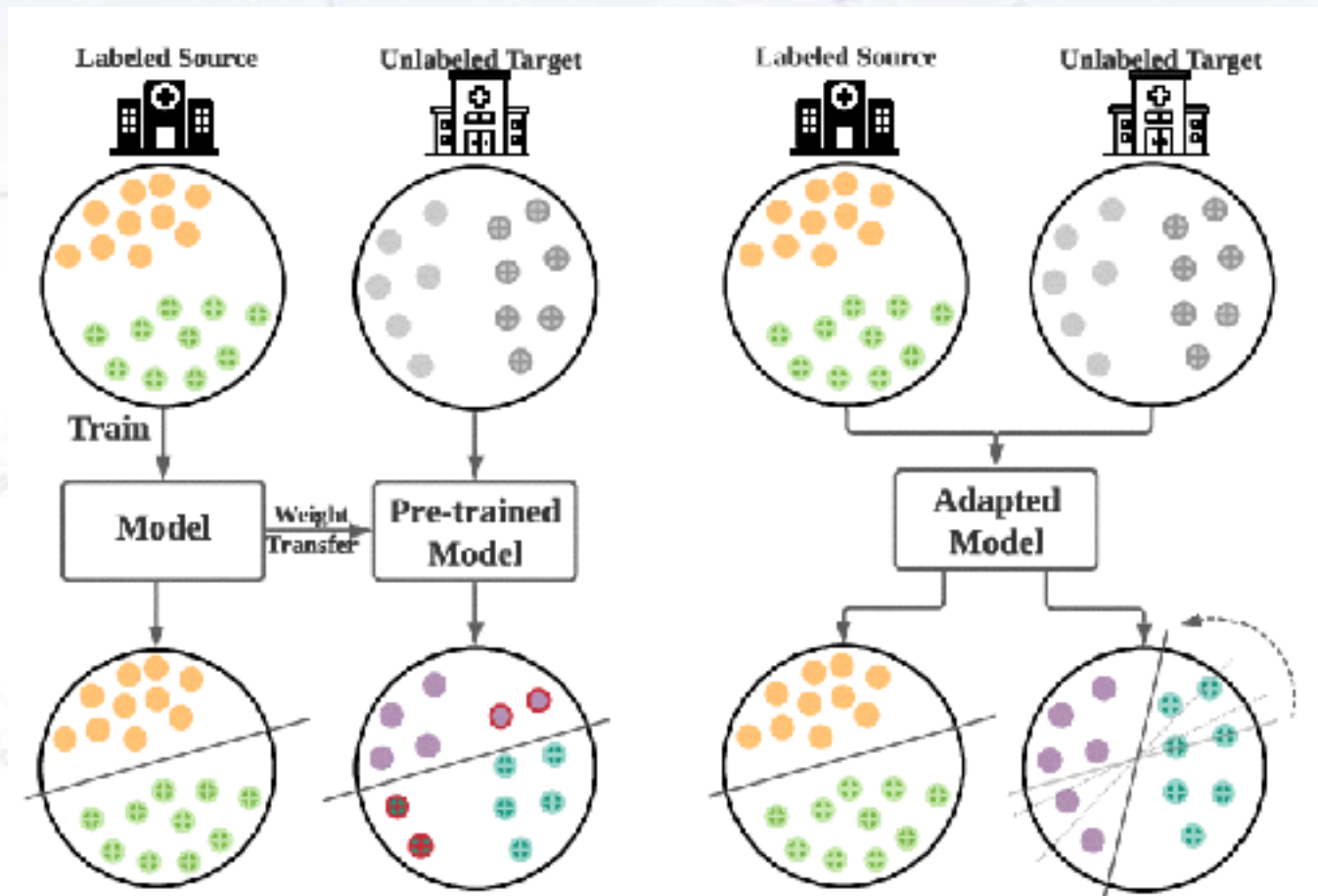


Domain Adaptation

Domain shift dangers

Domain shift refers to training an ML algorithm on one set of data (domain) and then applying it on another **different** data set (shift in domain).

Domain adaptation is ensuring that the trained model works on new data.



Domain shift dangers

The dangers are, that one does not know of or notice the domain shifts!

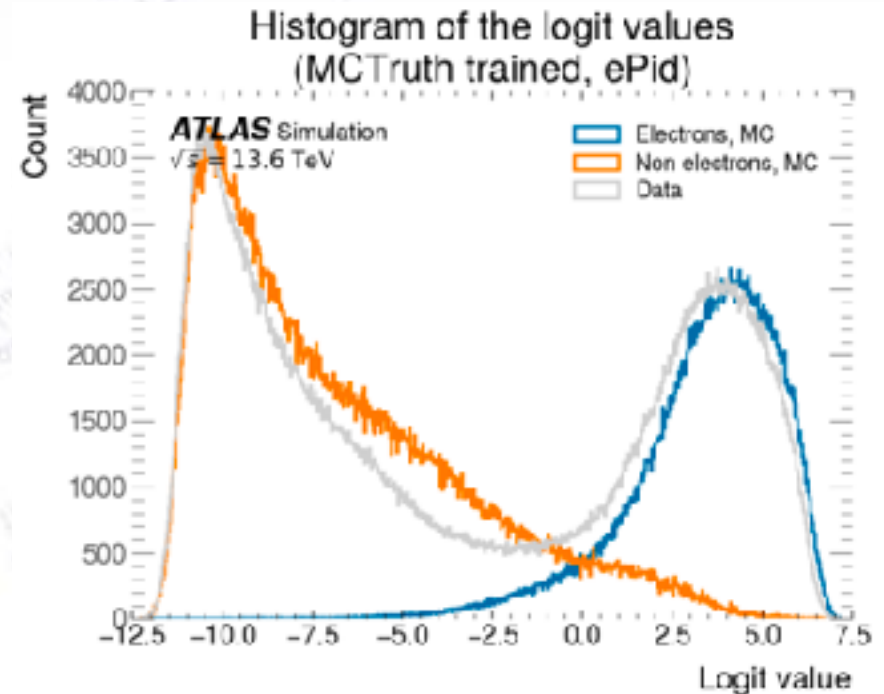
In order not to become a “domain shift victim”, one should:

- Plot the input variables for MC signal and background, with data on top, as shown below (though for ML output).
- If possible, use a Tag&Probe technique to get the signal and background distributions in data, and compare these (possibly also correlation matrix).

However, even if all of these are on top of each other, then there is no guarantee that data and MC is alike.

Simulating from first principle is what makes us hopeful that it is.

The real test is to see, if you can classify between data and MC!



Domain shift dangers

How about getting signal from MC but background from data? Then the algorithm should become good at rejecting background as it looks in data.

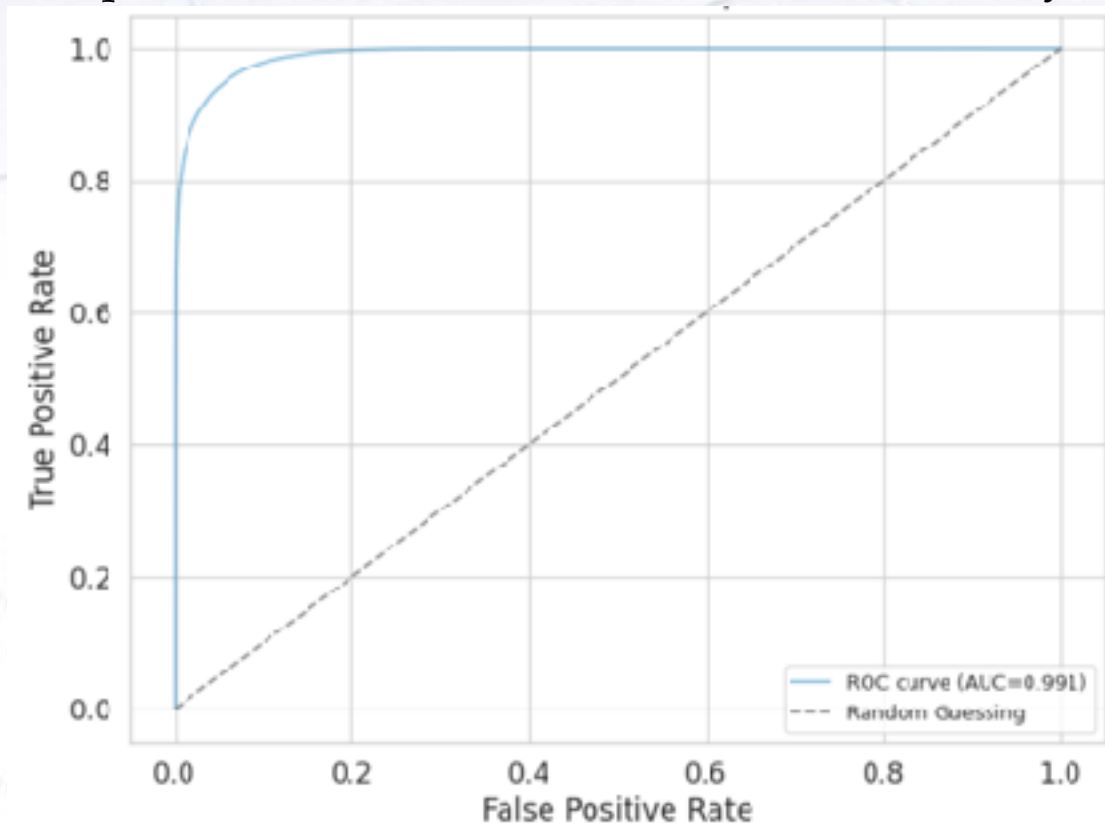
Well, not only that. If it is possible for the model to (also) detect differences between data and MC, then it partially learns to reject events that looks like data.

Thus, when employed on real data, it is less effective even to signal!

Finding data-MC differences

How to find differences between data and MC? The usual answer: Use ML!
More specifically, find a clean and relevant control sample, and compare.

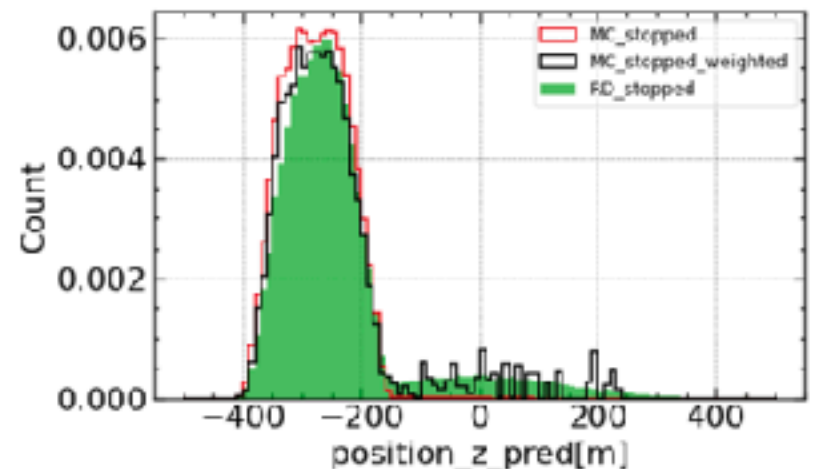
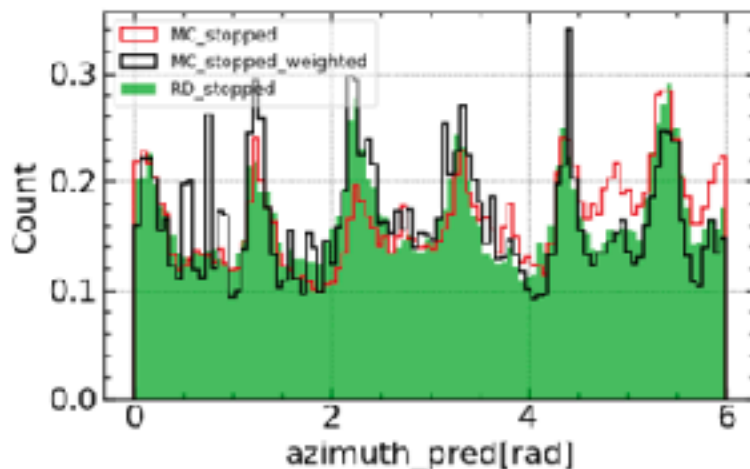
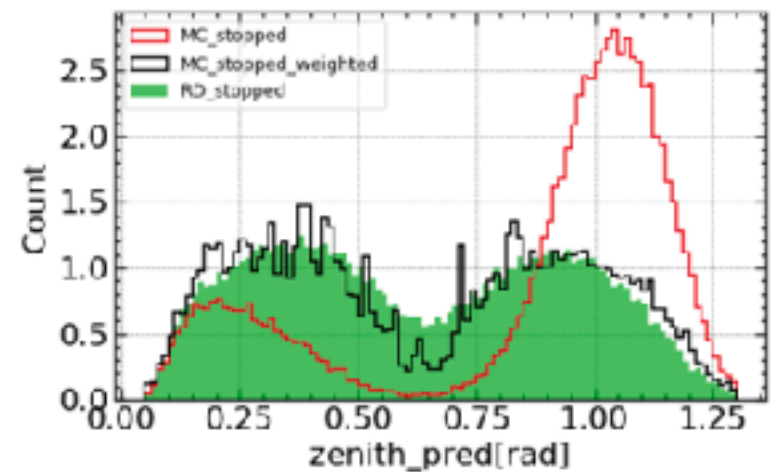
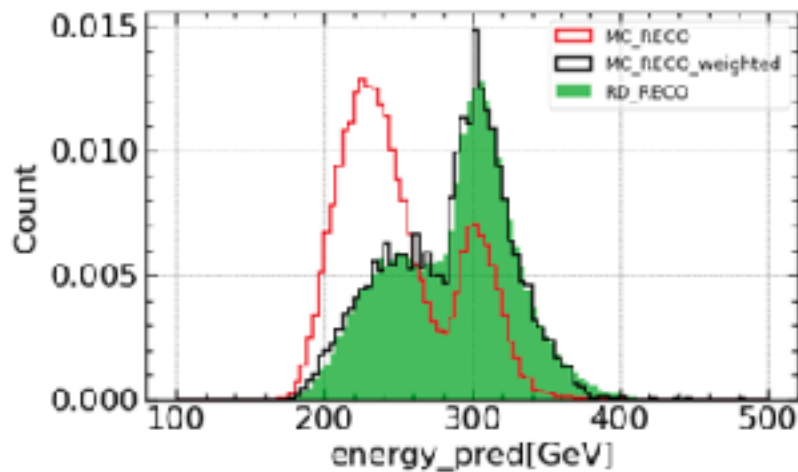
Here is an example (from cosmic muons in IceCube)... clearly different!



But are the differences related to learning to ID and reconstruct these muons?

Mending data-MC differences

The first step is to **reweights MC to match data** in the most important features, such as energy and direction. How? Use ML (GBReweigher).



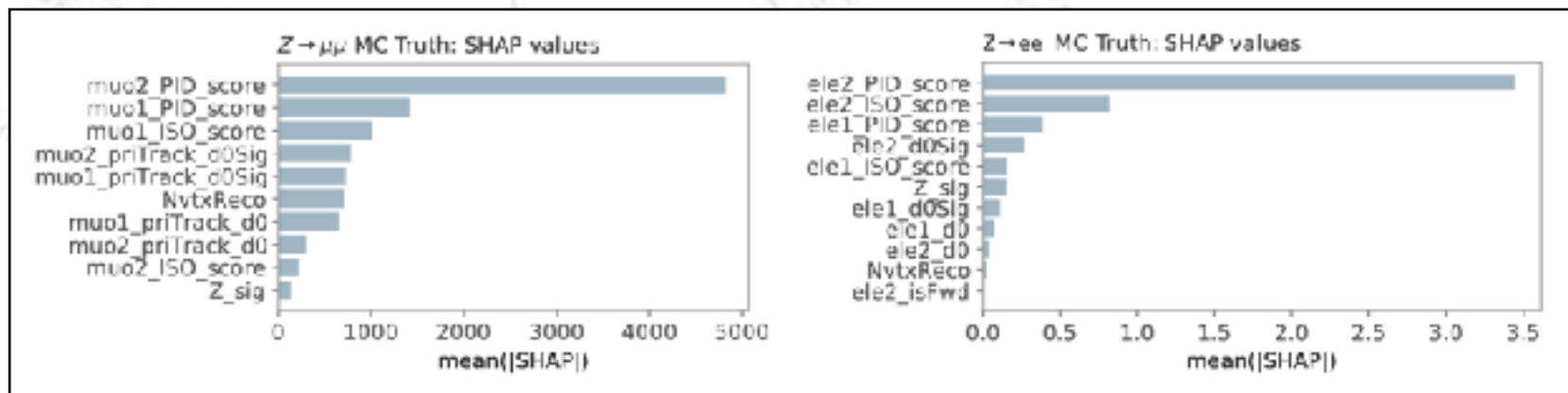
Mending data-MC differences

The next step is to check which variables are the most “guilty” in the differences, given by the feature ranking of the data-MC ML classifier.

Recall, this can be done in many ways:

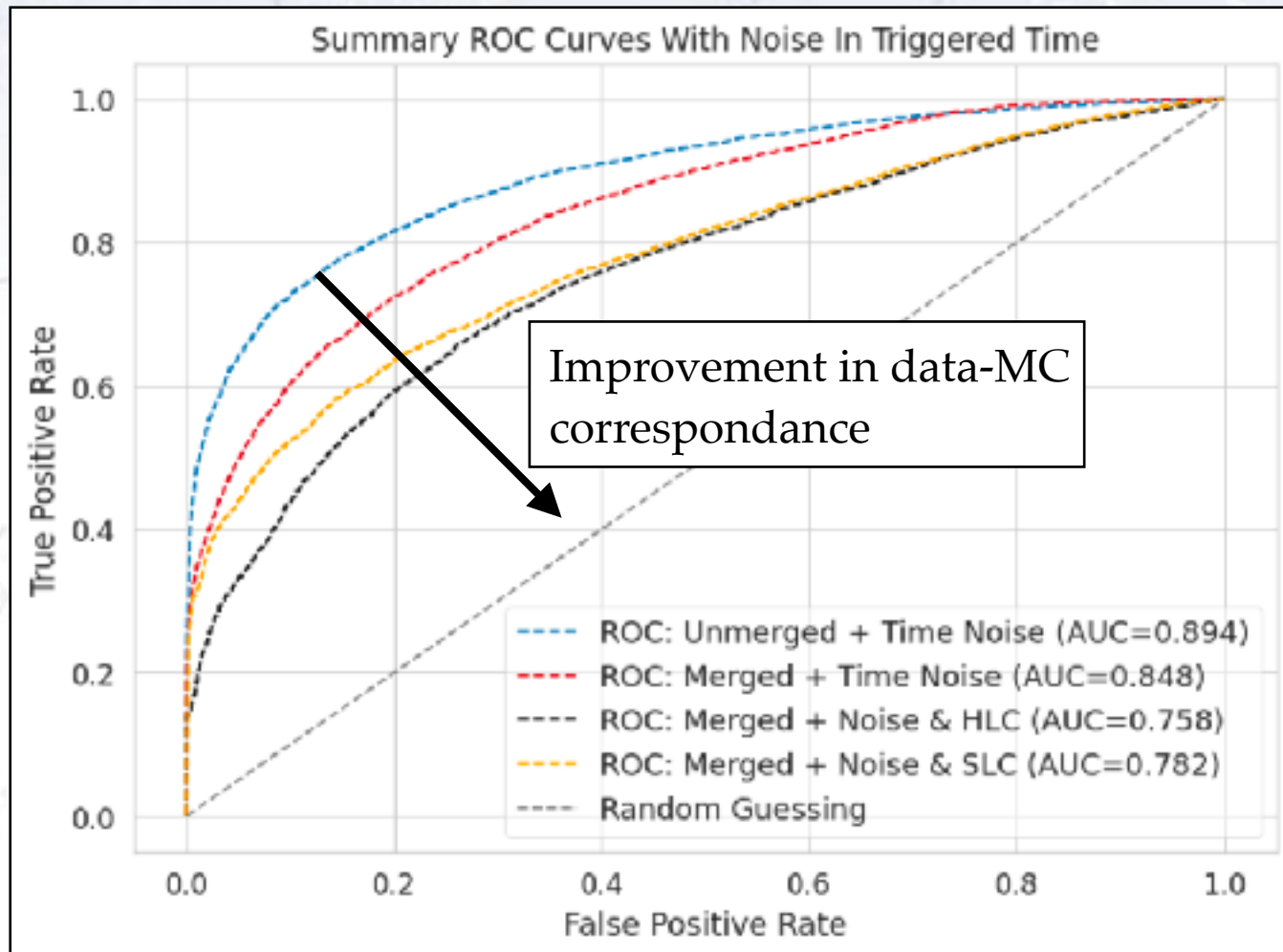
- SHAP values (also giving individual scores)
- Permutation invariance
- Build in methods

From such a ranking, one can see which variables to inspect. The challenge is, that if all 1D distributions match, then the corrections need to be done in higher dimensions!



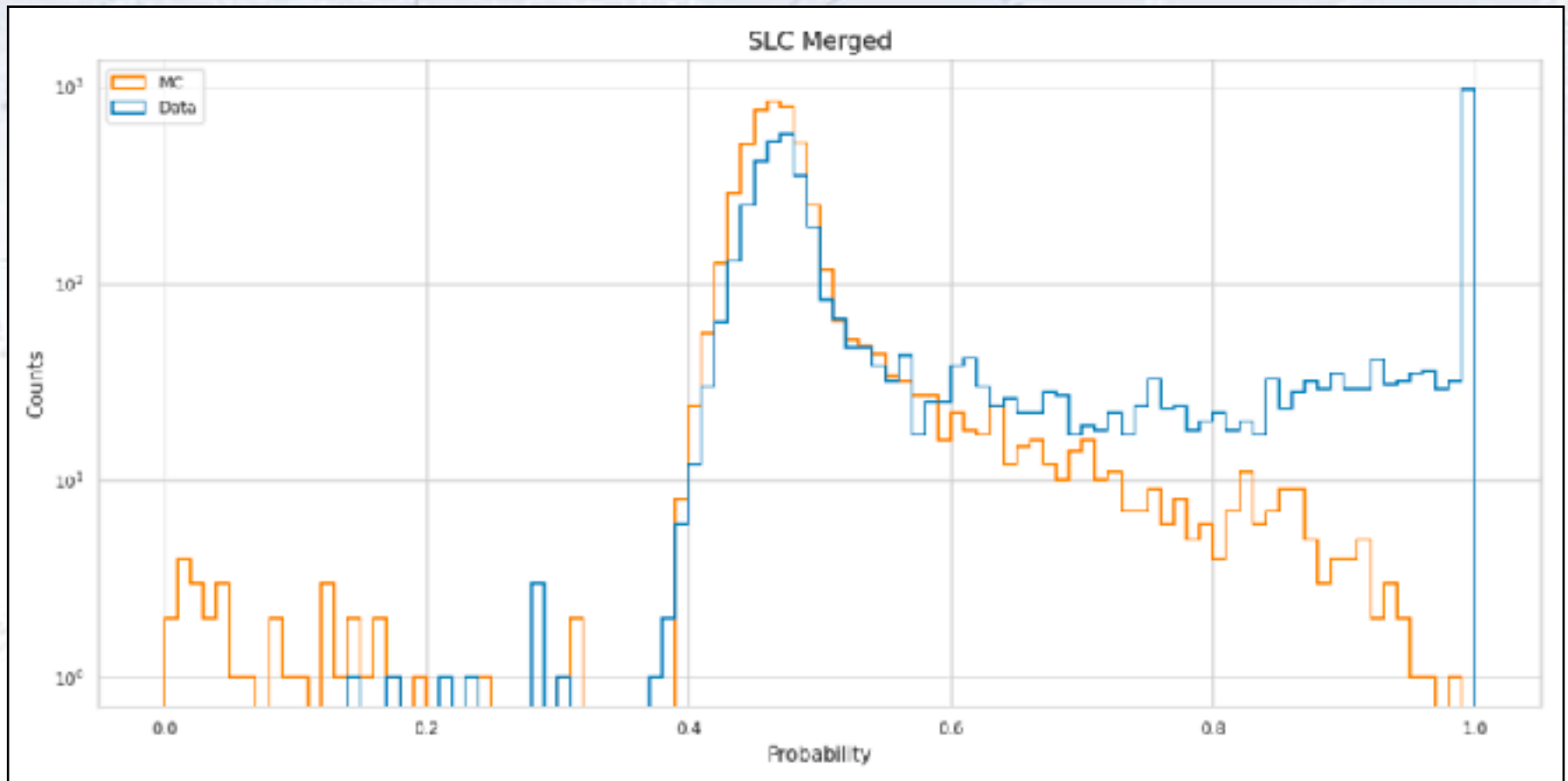
Mending data-MC differences

This is all hard work, but consider ML a helper to speed up the process (compared to earlier days). It is somehow the “inverse ROC game”.



Mending data-MC differences

At some point (well, the end), we reached this stage:



What do you conclude?

Training in/with data

Is it possible to train models on the real data? In particle physics, the answer is often “yes”, though it is rarely easy.

Think about how we calibrate/check our PID/reconstruction. We typically use a control channel with a known decay and resulting mass peak.

This can be used in a “reverse” manner: Given the known PID/energy of particles in real data, train on these!

The labels are of course approximate and the training sample finite (small?). This can be mitigated using “hybrid training” (yet to be published):

Train on data and MC simultaneously:

$$\mathcal{L} = \mathcal{L}_{\text{Data}} + \mathcal{L}_{\text{MC}} = \sum_{i \in \text{Data}} \mathcal{L}(\hat{y}_{\text{Data},i} - y_{\text{Data},i}) + \sum_{i \in \text{MC}} \mathcal{L}(\hat{y}_{\text{MC},i} - y_{\text{MC},i})$$

A faded nautical chart serves as the background. It features depth contours in fathoms, with labels such as 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200, 210, 220, 230, 240, 250, 260, 270, 280, 290, 300, 310, 320, 330, 340, 350, 360, 370, 380, 390, 400, 410, 420, 430, 440, 450, 460, 470, 480, 490, 500, 510, 520, 530, 540, 550, 560, 570, 580, 590, 600, 610, 620, 630, 640, 650, 660, 670, 680, 690, 700, 710, 720, 730, 740, 750, 760, 770, 780, 790, 800, 810, 820, 830, 840, 850, 860, 870, 880, 890, 900, 910, 920, 930, 940, 950, 960, 970, 980, 990, 1000. A red crosshair is centered on the chart, with a red 'X' at its intersection. The text 'W 51 10' 15' W' is visible near the crosshair. Other text on the chart includes 'PACIFIC' and '1871 BITTEN ERW TAUCHT 1875'.

When to apply ML?

When to use ML in HEP?

Using ML in an analysis is usually a (favorable) trade-off between:

- **Higher statistics** → **Lower statistical error**
(better efficiency, sharper peaks... *unless the cases are simple!*)
- **Larger data-MC differences** → **Higher systematic errors**
(more inputs, non-linearities... *unless there are good control channels!*)

So consider the table of uncertainties from a previous analysis (or estimate these with a colleague), and **ask yourself which of the two are dominant?**

Jet multiplicity	Measured cross section ± (stat.) ± (syst.) ± (lumi.) [pb]			
	$Z \rightarrow \ell\ell$			
≥ 0 jets	740 ±	1 ±	23 ±	16
≥ 1 jets	116.0 ±	0.3 ±	9.7 ±	2.5
≥ 2 jets	27.0 ±	0.1 ±	2.8 ±	0.6
≥ 3 jets	6.20 ±	0.04 ±	0.82 ±	0.14
≥ 4 jets	1.48 ±	0.02 ±	0.23 ±	0.04
≥ 5 jets	0.36 ±	0.01 ±	0.07 ±	0.01
≥ 6 jets	0.079 ±	0.004 ±	0.018 ±	0.002
≥ 7 jets	0.0178 ±	0.0019 ±	0.0049 ±	0.0005

Search for Higgs boson decays to a Z boson and a photon in proton-proton collisions at $\sqrt{s} = 13$ TeV

CMS Collaboration

Summary

A search is performed for a standard model (SM) Higgs boson decaying into a lepton pair (e^+e^- or $\mu^+\mu^-$) and a photon with $m_{\gamma,\ell} > 50$ GeV. The analysis is performed using a sample of proton-proton (pp) collision data at $\sqrt{s} = 13$ TeV, corresponding to an integrated luminosity of 138 fb⁻¹. The main contribution to this final state is from Higgs boson decays to a Z boson and a photon ($H \rightarrow Z\gamma \rightarrow \ell^+\ell^-\gamma$). The best fit value of the signal strength μ for $m_H = 125.38$ GeV is $\mu = 2.4^{+0.5}_{-0.9}$ (stat) $^{+0.3}_{-0.2}$ (syst) = 2.4 ± 0.6 . This measurement corresponds to $\sigma(pp \rightarrow H)B(H \rightarrow Z\gamma) = 0.21 \pm 0.09$ pb. The measured value is 1.6 standard deviations higher than the SM prediction. The observed (expected) local significance is 2.7 (1.2) standard deviations, where the expected significance is determined for the SM hypothesis. The observed (expected) upper limit at 95% confidence level on μ is 4.1 (1.6). In addition, a combined fit with the $H \rightarrow \gamma\gamma$ analysis of the same data set [18] is performed to measure the ratio $B(H \rightarrow Z\gamma)/B(H \rightarrow \gamma\gamma) = 1.5^{+1.1}_{-1.0}$, which is consistent with the ratio of 0.69 = 0.04 predicted by the SM at the 1.5 standard deviation level.

With this in mind, consider if it is worthwhile to apply Machine Learning.

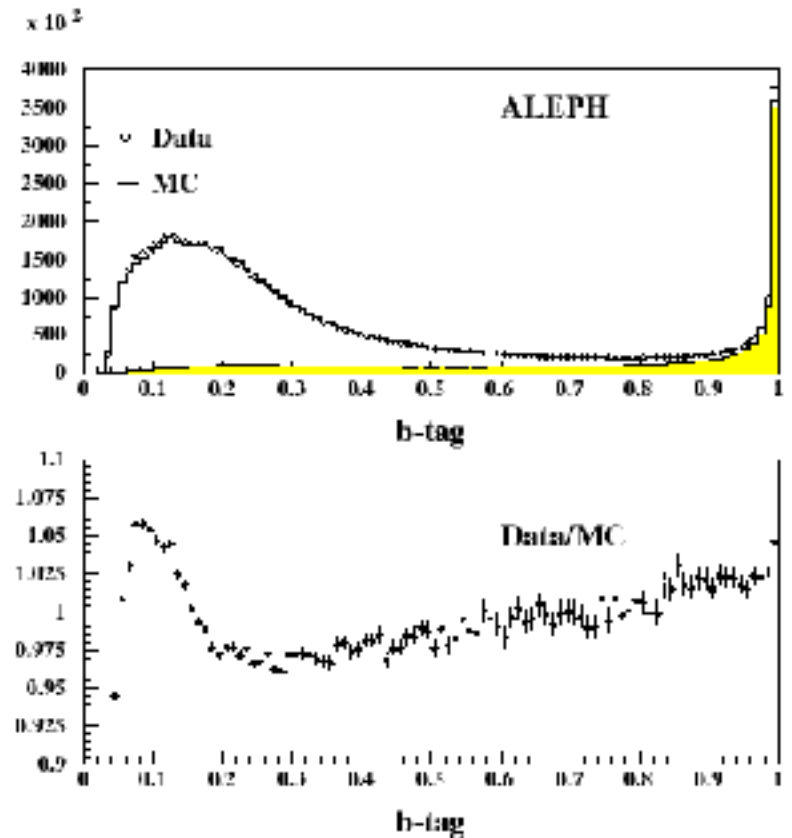
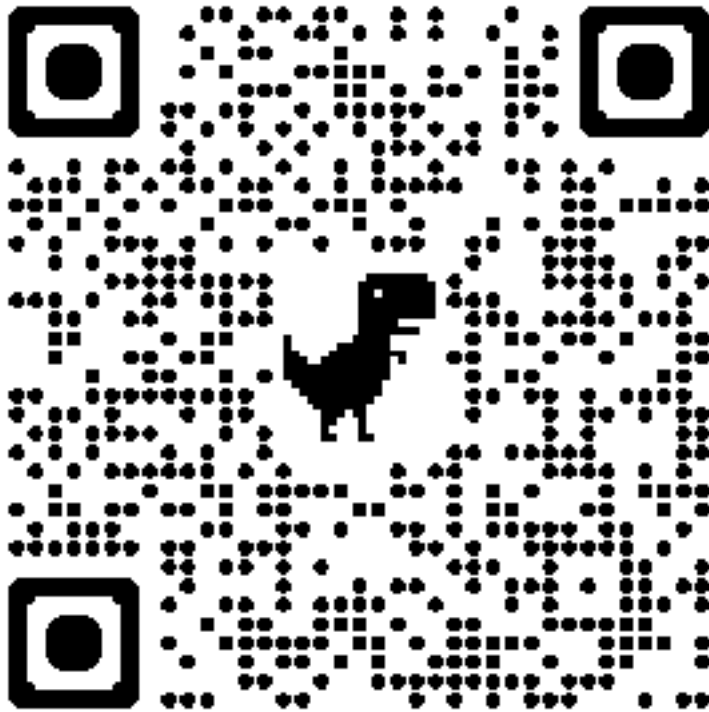


Coding examples

Example analysis

I've produced a HEP example of classification based on the Aleph data from LEP times (with BDTs and NNs applied).

It runs out of the box, and you are welcome to copy it for your own use :-)



https://github.com/troelspetersen/CERNSchoolOfPhysics_ML2024

The background is a bathymetric map of the Pacific Ocean. It features depth contours in fathoms, with labels such as 100, 150, 200, 250, 300, and 350. A prominent red shaded area is visible in the upper right quadrant, likely representing a specific oceanographic feature or data set. The map includes a coordinate grid with latitude and longitude lines. A specific location is marked with a cross and labeled 'VAR 10°13'W'. In the top right corner, there is a stamp that reads 'LIBRARY OF THE U.S. NAVY' and 'NAO 111/123'.

Summary

Summary

The main ingredients in ML are:

- Solutions exist (Universal Approximation Theorems)
- Solutions can be found (Stochastic Gradient Descent)
- Algorithms that are implemented:
 - Boosted Decision Trees
 - Neural Networks
- Knowledge about how to tell them what to learn (Loss function)
- A scheme for how to use the data (Splitting / Cross Validation)

When applying ML to HEP data, there are several challenges:

- Data and MC do not follow the same distributions!!!
 - Sometimes it is clear: Spells disaster
 - But when it is not clear, then the impact is unknown.
 - Therefore, always think in terms of control channels.
- Loss functions are important
 - They are your way of telling the algorithm what to do (i.e. optimise for).
- Training is great, but stopping when it is done is also important.
- Good control of dividing dataset is important
 - Use Cross Validation (CV) when there is little data or errors are needed.
- Print and plot your data as the first thing!
- **Work hard on getting MC to match data.**



Bonus slides

1987: First ML in HEP?

While this is incredibly ahead of its time, it also shows the sign of being early...

**NEURAL NETWORKS AND CELLULAR AUTOMATA
IN EXPERIMENTAL HIGH ENERGY PHYSICS**

B. DENBY
Laboratoire de l'Accélérateur Linéaire, Orsay, France

Received 26 September 1987; in revised form 28 December 1987

Within the past few years, two novel computing techniques, cellular automata and neural networks, have shown considerable promise in the solution of problems of a very high degree of complexity, such as turbulent fluid flow, image processing, and pattern recognition. Many of the problems faced in experimental high energy physics are also of this nature. Track reconstruction in wire chambers and cluster finding in cellular calorimeters, for instance, involve pattern recognition and high combinatorial complexity since many combinations of hits or cells must be considered in order to arrive at the final tracks or clusters. Here we evaluate in what way connective network methods can be applied to some of the problems of experimental high energy physics. It is found that such problems as track and cluster finding adapt naturally to these approaches. When large scale hard-wired connective networks become available, it will be possible to realize solutions to such problems in a fraction of the time required by traditional methods. For certain types of problems, faster solutions are already possible using model networks implemented on vector or other massively parallel machines. It should also be possible, using existing technology, to build simplified networks that will allow detailed reconstructed event information to be used in fast trigger decisions.

On the right is a figure showing a possible neural network implementation in hardware!

While seemingly premature, this might be the first FPGA design envisioned!

